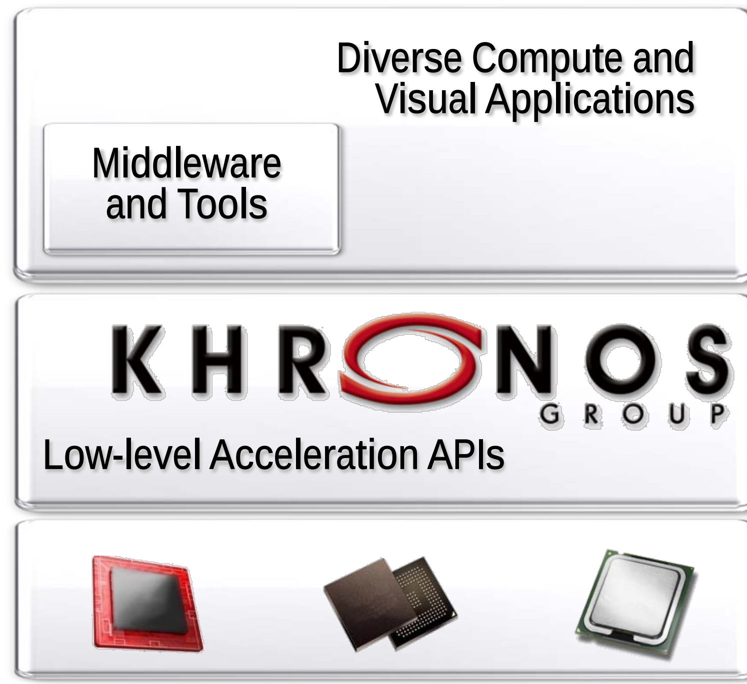
OpenCL

The Open Standard for Heterogeneous Parallel Programming

March 2009

“Close-to-the-Silicon” Standards

- Khronos creates “Foundation-Level” acceleration APIs
  - Needed on every platform to support an ecosystem of middleware and applications
- Low-level access to processor silicon
  - Designed with strong silicon vendor participation
- Cross-vendor software portability
  - API abstractions just high enough to hide implementation specifics
- Khronos has an established focus on graphics/media
  - 3D, vector 2D, video, imaging, audio APIs…
- …OpenCL broadens focus to Compute
  - Enabling applications to access the power of heterogeneous parallel computing silicon



Khronos APIs create the foundation of an ecosystem that enable applications to be PORTABLE and ACCELERATED on diverse silicon platforms

KHRONOS GROUP

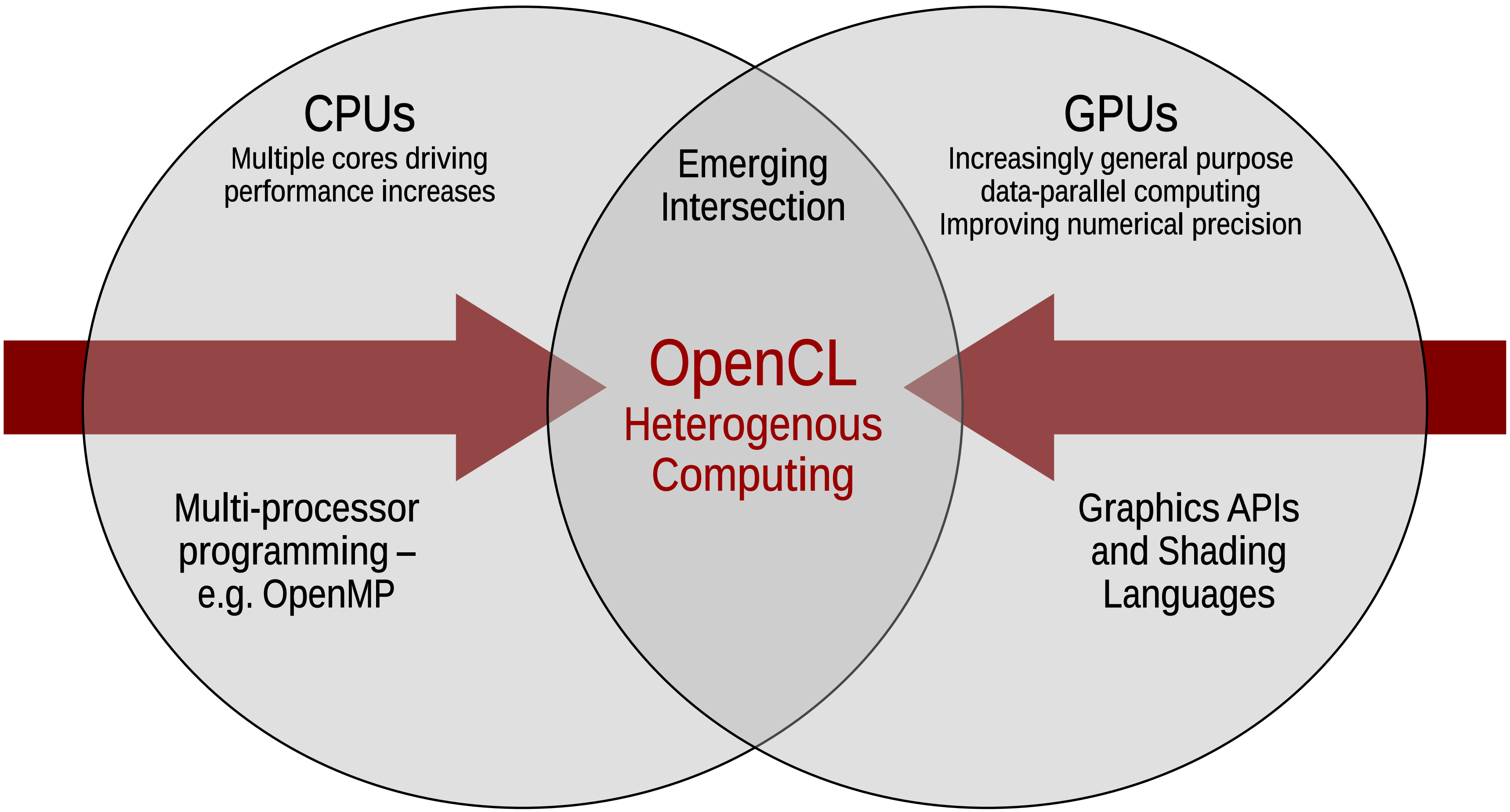
Over 100 companies creating authoring and acceleration standards

Board of Promoters

OpenCL Commercial Objectives

- Grow the market for parallel computing
  - For vendors of systems, silicon, middleware, tools and applications
- Open, royalty-free standard for heterogeneous parallel computing
  - Unified programming model for CPUs, GPUs, Cell, DSP and other processors in a system
- Cross-vendor software portability to a wide range of silicon and systems
  - HPC servers, desktop systems and handheld devices covered in one specification
- Support for a wide diversity of applications
  - From embedded and mobile software through consumer applications to HPC solutions
- Create a foundation layer for a parallel computing ecosystem
  - Close-to-the-metal interface to support a rich diversity of middleware and applications
- Rapid deployment in the market
  - Designed to run on current latest generations of GPU hardware

Processor Parallelism



OpenCL – Open Computing Language

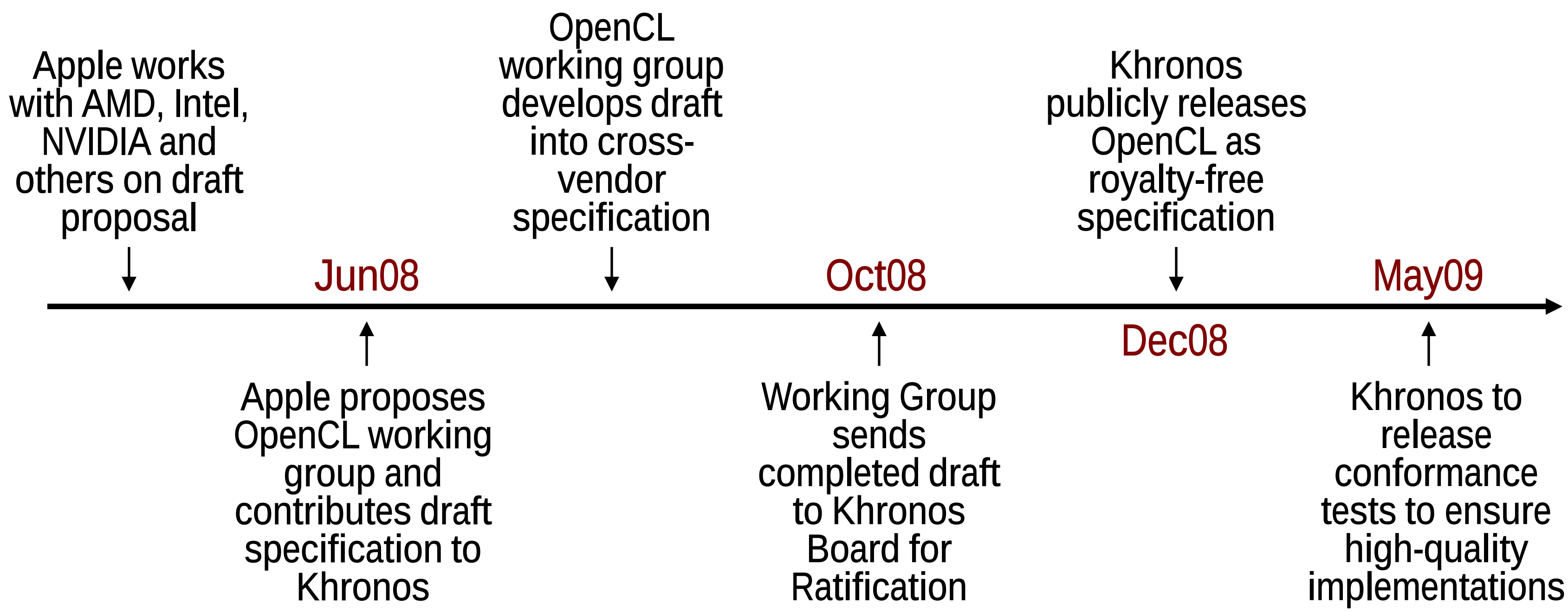
Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

OpenCL Working Group

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers
- Many industry-leading experts involved in OpenCL's design
  - A healthy diversity of industry perspectives
- Apple initially proposed and is very active in the working group
  - Serving as specification editor
- Here are some of the other companies in the OpenCL working group

OpenCL Timeline

- Six months from proposal to released specification
  - Due to a strong initial proposal and a shared commercial incentive to work quickly
- Apple's Mac OS X Snow Leopard will include OpenCL
  - Improving speed and responsiveness for a wide spectrum of applications
- Multiple OpenCL implementations expected in the next 12 months
  - On diverse platforms



What Does This Mean to Me?

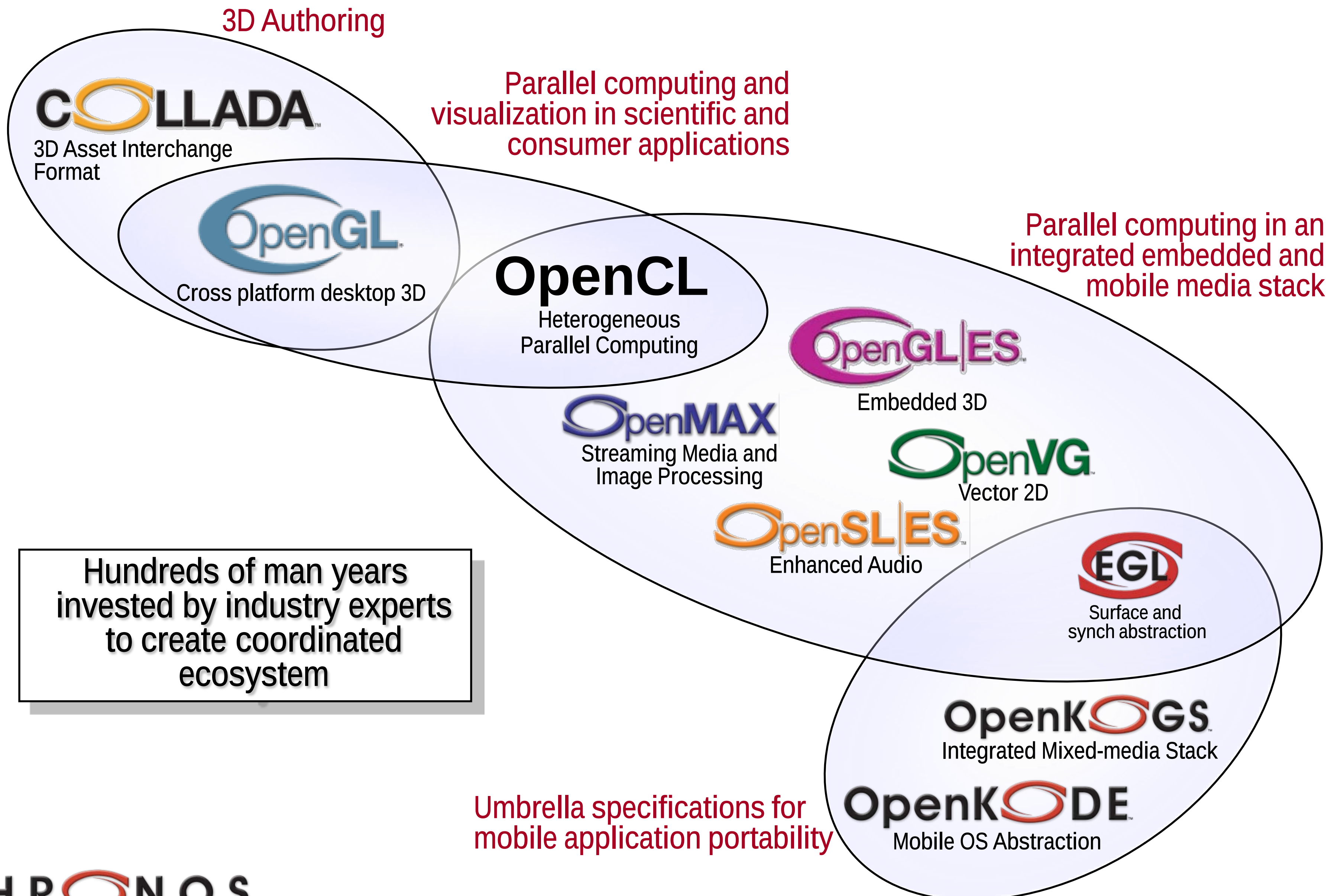
- Software developers
  - OpenCL enables you to write parallel programs that will run portably on many devices
  - Royalty-free – with no cost to use the API
  - Soon - should have a wider choice of compute tools, libraries and middleware
- Silicon and hardware vendors
  - OpenCL lets you tap into the building momentum of OpenCL applications and middleware
  - The specification is available free of charge to use on the Khronos web-site
  - Conformance Tests and the OpenCL Adopters Program ready in spring
- OpenCL implementations must pass conformance tests to use trademark
  - Khronos will license tests for nominal fee to any interested company
- .. and most importantly - end-users will benefit
  - A wide range of innovative applications will be enabled and accelerated by unleashing the parallel computing capabilities of their systems and devices

OpenCL 1.0 Embedded Profile

- Enables OpenCL on mobile and embedded silicon
  - Relaxes some data type and precision requirements
  - Avoids the need for a separate "ES" specification
- Khronos mobile API ecosystem defines mixed compute, imaging/ graphics
  - Enabling advanced applications e.g. augmented reality
- OpenCL will enable parallel computing in new market areas
  - E.g. mobile phones, automotive, avionics

A GPS phone processes images to recognize buildings and landmarks and uses the internet to supply relevant data

OpenCL and the Khronos Ecosystem



Hundreds of man years invested by industry experts to create coordinated ecosystem

OpenCL Technical Overview

OpenCL Design Requirements

- Use all computational resources in system
  - Program GPUs, CPUs, and other processors as peers
  - Support both data- and task- parallel compute models
- Efficient C-based parallel programming model
  - Abstract the specifics of underlying hardware
- Abstraction is low-level, high-performance but device-portable
  - Approachable – but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or "convenience" functions
- Implementable on a range of embedded, desktop, and server systems
  - HPC, desktop, and handheld profiles in one specification
- Drive future hardware requirements
  - Floating point precision requirements
  - Applicable to both consumer and HPC applications

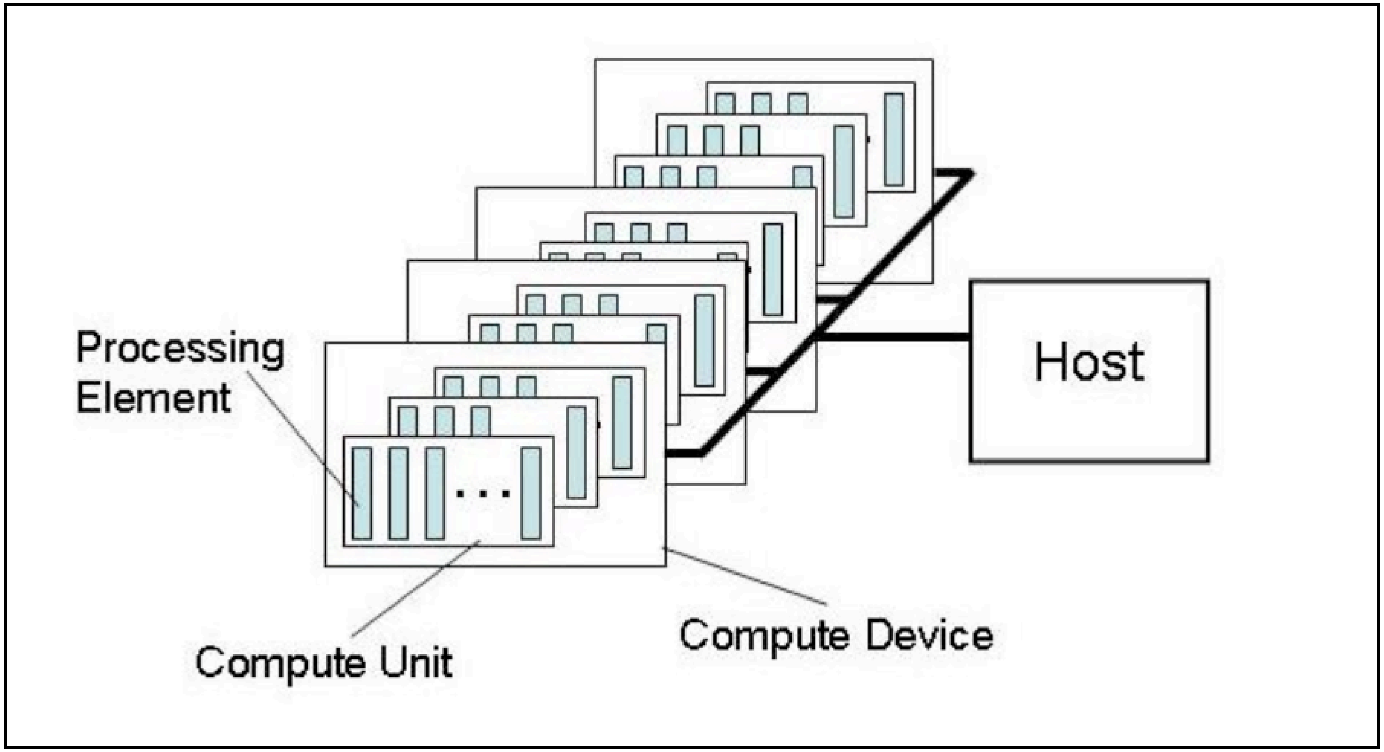
Anatomy of OpenCL

- Language Specification
  - C-based cross-platform programming interface
  - Subset of ISO C99 with language extensions - familiar to developers
  - Well-defined numerical accuracy - IEEE 754 rounding behavior with specified maximum error
  - Online or offline compilation and build of compute kernel executables
  - Includes a rich set of built-in functions
- Platform Layer API
  - A hardware abstraction layer over diverse computational resources
  - Query, select and initialize compute devices
  - Create compute contexts and work-queues
- Runtime API
  - Execute compute kernels
  - Manage scheduling, compute, and memory resources

Hierarchy of Models

- Platform Model
- Memory Model
- Execution Model
- Programming Model

OpenCL Platform Model (Section 3.1)

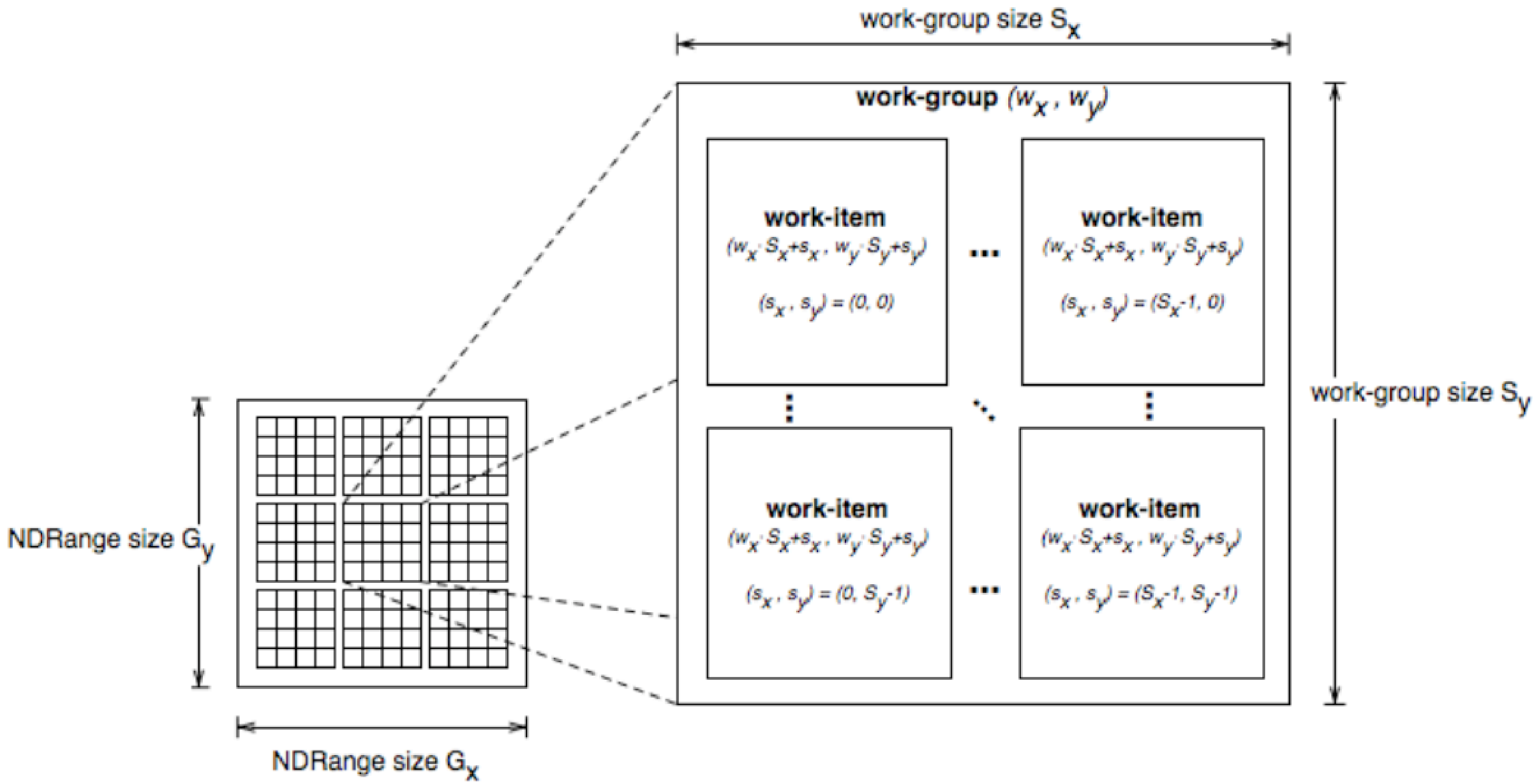


- One Host + one or more Compute Devices
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

OpenCL Execution Model (Section 3.2)

- OpenCL Program:
  - Kernels
    - Basic unit of executable code — similar to a C function
    - Data-parallel or task-parallel
  - Host Program
    - Collection of compute kernels and internal functions
    - Analogous to a dynamic library
- Kernel Execution
  - The host program invokes a kernel over an index space called an *NDRange*
    - NDRange = "N-Dimensional Range"
    - NDRange can be a 1, 2, or 3-dimensional space
  - A single kernel instance at a point in the index space is called a *work-item*
    - Work-items have unique global IDs from the index space
  - Work-items are further grouped into *work-groups*
    - Work-groups have a unique work-group ID
    - Work-items have a unique local ID within a work-group

Kernel Execution



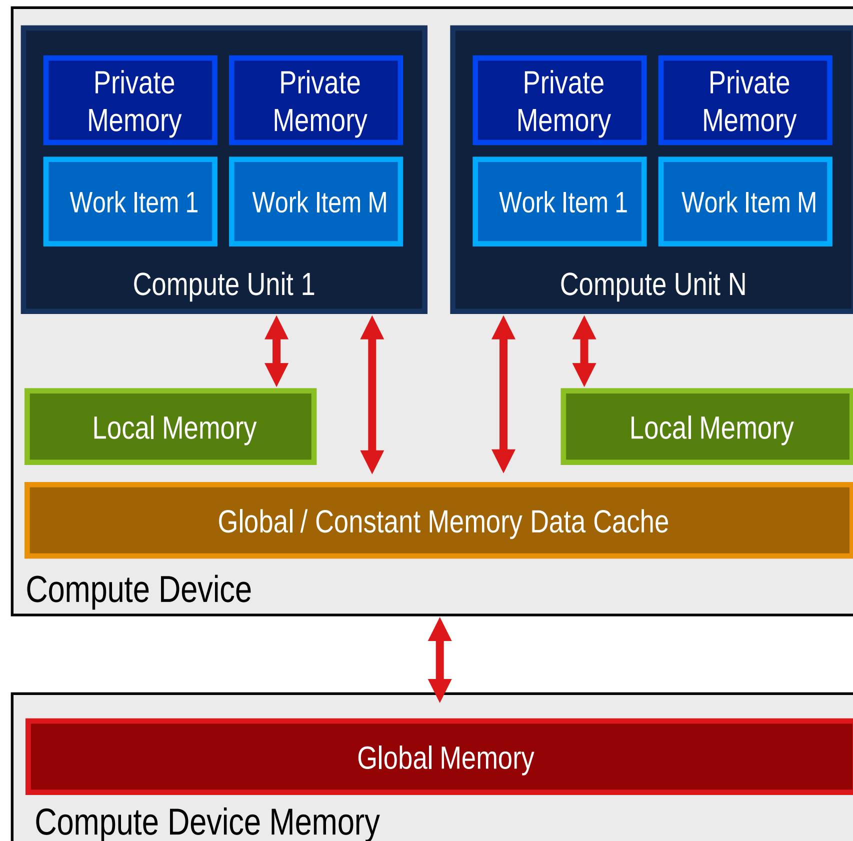
- Total number of work-items = G_x x G_y
- Size of each work-group = S_x x S_y
- Global ID can be computed from work-group ID and local ID

Contexts and Queues (Section 3.2.1)

- Contexts are used to contain and manage the state of the “world”
- Kernels are executed in contexts defined and manipulated by the host
  - Devices
  - Kernels - OpenCL functions
  - Program objects - kernel source and executable
  - Memory objects
- Command-queue - coordinates execution of kernels
  - Kernel execution commands
  - Memory commands - transfer or mapping of memory object data
  - Synchronization commands - constrains the order of commands
- Applications queue compute kernel execution instances
  - Queued in-order
  - Executed in-order or out-of-order
  - Events are used to implement appropriate synchronization of execution instances

OpenCL Memory Model (Section 3.3)

- Shared memory model
  - Relaxed consistency
- Multiple distinct address spaces
  - Address spaces can be collapsed depending on the device's memory subsystem
- Address spaces
  - Private - private to a *work-item*
  - Local - local to a *work-group*
  - Global - accessible by all work-items in all work-groups
  - Constant - read only global space
- Implementations map this hierarchy
  - To available physical memories



Memory Consistency (Section 3.3.1)

- "OpenCL uses a relaxed consistency memory model; i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times."
- Within a work-item, memory has load/store consistency
- Within a work-group at a barrier, local memory has consistency across work-items
- Global memory is consistent within a work-group, at a barrier, but not guaranteed across different work-groups
- Consistency of memory shared between commands are enforced through synchronization

Data-Parallel Programming Model
(Section 3.4.1)

- Define N-Dimensional computation domain
  - Each independent element of execution in an N-Dimensional domain is called a *work-item*
  - N-Dimensional domain defines the total number of work-items that execute in parallel = *global work size*
- Work-items can be grouped together — *work-group*
  - Work-items in group can communicate with each other
  - Can synchronize execution among work-items in group to coordinate memory access
- Execute multiple work-groups in parallel
  - Mapping of global work size to work-group can be implicit or explicit

Task-Parallel Programming Model

(Section 3.4.2)

- Data-parallel execution model must be implemented by all OpenCL compute devices
- Some compute devices such as CPUs can also execute task-parallel compute kernels
  - Executes as a single work-item
  - A compute kernel written in OpenCL
  - A native C / C++ function

Basic OpenCL Program Structure

- Host program
  - Query compute devices
  - Create contexts
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources
- Kernels
  - C code with some restrictions and extensions

Platform Layer

Runtime

Language

Example: Vector Addition

- Compute c = a + b
  - a, b, and c are vectors of length *N*

- Basic OpenCL concepts
  - Simple kernel code
  - Basic context management
  - Memory allocation
  - Kernel invocation

Platform Layer (Chapter 4)

- Platform layer allows applications to query for platform specific features
- Querying platform info (i.e., OpenCL profile) (Chapter 4.1)
- Querying devices (Chapter 4.2)
  - *clGetDeviceIDs*()
    - Find out what compute devices are on the system
    - Device types include CPUs, GPUs, or Accelerators
  - *clGetDeviceInfo*()
    - Queries the capabilities of the discovered compute devices such as:
      - Number of compute cores
      - NDRange limits
      - Maximum work-group size
      - Sizes of the different memory spaces (constant, local, global)
      - Maximum memory object size
- Creating contexts (Chapter 4.3)
  - Contexts are used by the OpenCL runtime to manage objects and execute kernels on one or more devices
  - Contexts are associated to one or more devices
    - Multiple contexts could be associated to the same device
  - *clCreateContext*() and *clCreateContextFromType*() returns a *handle* to the created contexts

Command-Queues (Section 5.1)

- Command-queues store a set of operations to perform
- Command-queues are associated to a context
- Multiple command-queues can be created to handle independent commands that don't require synchronization
- Execution of the command-queue is guaranteed to be completed at sync points

VecAdd: Context, Devices, Queue

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,     // (must be 0)
                                             CL_DEVICE_TYPE_GPU,
                                             NULL,  // error callback
                                             NULL,  // user data
                                             NULL); // error code

// get the list of GPU devices associated with context
size_t cb;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cl_cmd_queue cmd_queue = clCreateCommandQueue(context,
                                              devices[0],
                                              0,     // default options
                                              NULL); // error code
```

Spec

Contexts and context creation:	Section 4.3
Command Queues:	Section 5.1

Memory Objects (Section 5.2)

- Buffer objects
  - One-dimensional collection of objects (like C arrays)
  - Valid elements include scalar and vector types as well as user defined structures
  - Buffer objects can be accessed via pointers in the kernel
- Image objects
  - Two- or three-dimensional texture, frame-buffer, or images
  - Must be addressed through built-in functions
- Sampler objects
  - Describes how to sample an image in the kernel
    - Addressing modes
    - Filtering modes

Creating Memory Objects

- *clCreateBuffer()*, *clCreateImage2D*(), and *clCreateImage3D*()
- Memory objects are created with an associated context
- Memory can be created as read only, write only, or read-write
- Where objects are created in the platform memory space can be controlled
  - Device memory
  - Device memory with data copied from a host pointer
  - Host memory
  - Host memory associated with a pointer
    - Memory at that pointer is guaranteed to be valid at synchronization points
- Image objects are also created with a channel format
  - Channel order (e.g., RGB, RGBA ,etc.)
  - Channel type (e.g., UNORM INT8, FLOAT, etc.)

Manipulating Object Data

- Object data can be copied to host memory, from host memory, or to other objects
- Memory commands are enqueued in the command buffer and processed when the command is executed
  - *clEnqueueReadBuffer*(), *clEnqueueReadImage*()
  - *clEnqueueWriteBuffer*(), *clEnqueueWriteImage*()
  - *clEnqueueCopyBuffer*(), *clEnqueueCopyImage*()
- Data can be copied between Image and Buffer objects
  - *clEnqueueCopyImageToBuffer*()
  - *clEnqueueCopyBufferToImage*()
- Regions of the object data can be accessed by mapping into the host address space
  - *clEnqueueMapBuffer*(), *clEnqueueMapImage*()
  - *clEnqueueUnmapMemObject*()

VecAdd: Create Memory Objects

```
cl_mem memobjs[3];

// allocate input buffer memory objects
memobjs[0] = clCreateBuffer(context,
                            CL_MEM_READ_ONLY |    // flags
                            CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float)*n,   // size
                            srcA,                 // host pointer
                            NULL);                // error code

memobjs[1] = clCreateBuffer(context,
                            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float)*n, srcB, NULL);

// allocate input buffer memory object
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);
```

Spec

Creating buffer objects: Section 5.2.1

Program Objects (Section 5.4)

- Program objects encapsulate:
  - An associated context
  - Program source or binary
  - Latest successful program build, list of targeted devices, build options
  - Number of attached kernel objects
- Build process
  1. Create program object
     - clCreateProgramWithSource()
     - clCreateProgramWithBinary()
  2. Build program executable
     - Compile and link from source or binary for all devices or specific devices in the associated context
     - clBuildProgram()
     - Build options
       - Preprocessor
       - Math intrinsics (floating-point behavior)
       - Optimizations

Kernel Objects (Section 5.5)

- Kernel objects encapsulate
  - Specific kernel functions declared in a program
  - Argument values used for kernel execution
- Creating kernel objects
  - *clCreateKernel*() - creates a kernel object for a single function in a program
  - *clCreateKernelsInProgram*() - creates an object for all kernels in a program
- Setting arguments
  - *clSetKernelArg*(<kernel>, <argument index>)
  - Each argument data must be set for the kernel function
  - Argument values are copied and stored in the kernel object
- Kernel vs. program objects
  - Kernels are related to program execution
  - Programs are related to program source

VecAdd: Program and Kernel

```
// create the program
cl_program program = clCreateProgramWithSource(
                                   context,
                                   1,               // string count
                                   &program_source, // program strings
                                   NULL,            // string lengths
                                   NULL);           // error code
// build the program
cl_int err = clBuildProgram(program,
                            0,     // num devices in device list
                            NULL,  // device list
                            NULL,  // options
                            NULL,  // notifier callback function ptr
                            NULL); // user data
// create the kernel
cl_kernel kernel = clCreateKernel(program, “vec_add”, NULL);
```

Spec		
	Creating program objects:	Section 5.4.1
	Building program executables:	Section 5.4.2
	Creating kernel objects:	Section 5.5.1

VecAdd: Set Kernel Arguments

```
// set “a” vector argument
err  = clSetKernelArg(kernel,
                      0,                   // argument index
                      (void *)&memobjs[0], // argument data
                      sizeof(cl_mem));     // argument data size

// set “b” vector argument
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem));

// set “c” vector argument
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2], sizeof(cl_mem));
```

Spec

Setting kernel arguments:	Section 5.5.2
Executing Kernels:	Section 6.1
Reading, writing, and copying buffer objects:	Section 5.2.2

KHRONOS GROUP

Kernel Execution (Section 5.6)

- A command to execute a kernel must be enqueued to the command-queue
- *clEnqueueNDRangeKernel*()
  - Data-parallel execution model
  - Describes the *index space* for kernel execution
  - Requires information on NDRange dimensions and work-group size
- *clEnqueueTask*()
  - Task-parallel execution model (multiple queued tasks)
  - Kernel is executed on a single work-item
- *clEnqueueNativeKernel*()
  - Task-parallel execution model
  - Executes a native C/C++ function not compiled using the OpenCL compiler
  - This mode does not use a kernel object so arguments must be passed in

Command-Queues and Synchronization

- Command-queue execution
  - Execution model signals when commands are complete or data is ready
  - Command-queue could be explicitly flushed to the device
  - Command-queues execute in-order or out-of-order
    - In-order - commands complete in the order queued and correct memory is consistent
    - Out-of-order - no guarantee when commands are executed or memory is consistent without synchronization
- Synchronization
  - Signals when commands are completed to the host or other commands in queue
  - Blocking calls
    - Commands that do not return until complete
    - clEnqueueReadBuffer() can be called as blocking and will block until complete
  - *Event objects*
    - Tracks execution status of a command
    - Some commands can be blocked until event objects signal a completion of previous command
      - clEnqueueNDRangeKernel() can take an event object as an argument and wait until a previous command (e.g., clEnqueueWriteBuffer) is complete
    - Profiling
  - Queue barriers - queued commands that can block command execution

VecAdd: Invoke Kernel, Read Output

```
size_t global_work_size[1] = n; // set work-item dimensions

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel,
                             1,          // Work dimensions
                             NULL,       // must be NULL (work offset)
                             global_work_size,
                             NULL,       // automatic local work size
                             0,          // no events to wait on
                             NULL,       // event list
                             NULL);      // event for this kernel
// read output array
err = clEnqueueReadBuffer(       context, memobjs[2],
                                 CL_TRUE,             // blocking
                                 0,                   // offset
                                 n*sizeof(cl_float),  // size
                                 dst,                 // pointer
                                 0, NULL, NULL);      // events
```

Spec

Setting kernel arguments:	Section 5.5.2
Executing Kernels:	Section 6.1
Reading, writing, and copying buffer objects:	Section 5.2.2

OpenCL C for Compute Kernels
(Chapter 6)

- Derived from ISO C99
  - A few restrictions: recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported
- Built-in Data Types
  - Scalar and vector data types, Pointers
  - Data-type conversion functions: convert_type<_sat><_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t
- Built-in Functions — Required
  - work-item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
- Built-in Functions — Optional
  - double precision, atomics to global and local memory
  - selection of rounding mode, writes to image3d_t surface

OpenCL C Language Highlights

- Function qualifiers
  - "__kernel" qualifier declares a function as a kernel
  - Kernels can call other kernel functions
- Address space qualifiers
  - __global, __local, __constant, __private
  - Pointer kernel arguments must be declared with an address space qualifier
- Work-item functions
  - Query work-item identifiers
  - get_work_dim()
  - get_global_id(), get_local_id(), get_group_id()
- Image functions
  - Images must be accessed through built-in functions
  - Reads/writes performed through sampler objects from host or defined in source
- Synchronization functions
  - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
  - Memory fences - provides ordering between memory operations

Vector Addition Kernel

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

Spec

`__kernel`:	Section 6.7.1
`__global`:	Section 6.5.1
`get_global_id()`:	Section 6.11.1
Data types:	Section 6.1

KHRONOS GROUP

OpenCL C Language Restrictions

- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported
- Writes to a pointer of types less than 32-bit are not supported
- Double types are not supported, but reserved
- 3D Image writes are not supported

- Some restrictions are addressed through extensions

Optional Extensions (Chapter 9)

- Extensions are optional features exposed through OpenCL
- The OpenCL working group has already approved many extensions that are supported by the OpenCL specification:
- Double precision floating-point types (Section 9.3)
- Built-in functions to support doubles
- Atomic functions (Section 9.5, 9.6, 9.7)
- 3D Image writes (Section 9.8)
- Byte addressable stores (write to pointers with types < 32-bits) (Section 9.9)
- Built-in functions to support half types (Section 9.10)

OpenGL Interoperability (Appendix B)

- Both standards under one IP framework
  - Enables very close collaborative design
- Efficient, inter-API communication
  - While still allowing both APIs to handle the types of workloads for which they were designed
- OpenCL can efficiently share resources with OpenGL
  - Textures, Buffer Objects and Renderbuffers
  - Data is shared, not copied
  - OpenCL objects are created *from* OpenGL objects
    - *clCreateFromGLBuffer*(), *clCreateFromGLTexture2D*(), *clCreateFromGLRenderbuffer*()
- Applications can select compute device(s) to run OpenGL and OpenCL
  - Efficient queuing of OpenCL and OpenGL commands into the hardware
  - Flexible scheduling and synchronization
- Examples
  - Vertex and image data generated with OpenCL and then rendered with OpenGL
  - Images rendered with OpenGL and post-processed with OpenCL kernels

OpenCL Summary

- Cross-vendor standard for portable heterogeneous programming
  - Open, royalty-free standard with critical mass support from key vendors
- Creates significant commercial opportunities
  - Removes fragmentation as market barrier to the growth of parallel computing
- A central role in the Khronos API Ecosystem
  - Multiple related APIs are being collaboratively developed under one IP framework at Khronos
- Fast track deployment
  - Public specification created in under 6 months - for implementations in 2009
  - Will run on current latest generations of GPU hardware
- The specification and these slides at www.khronos.org/opencl/
  - If this is relevant to your company – please join Khronos and get involved!