CSE 531

OpenCL

Spring 2023

Mahmut Taylan Kandemir

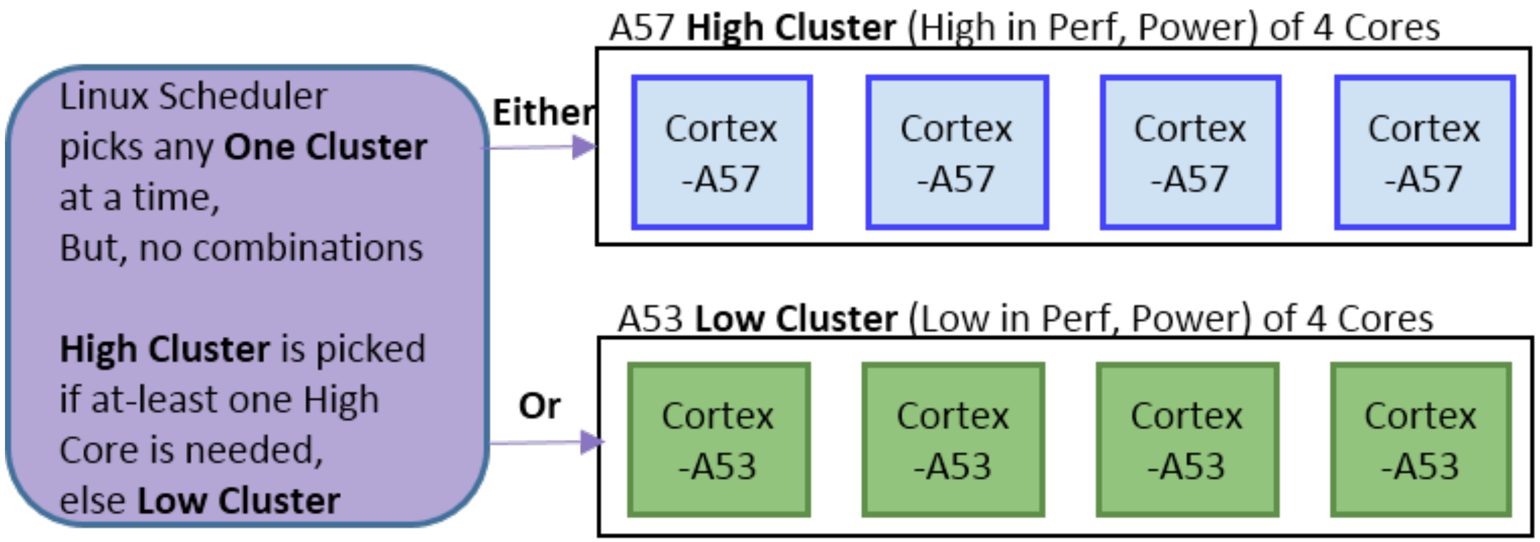
ACK: D. Exterman, B. Catanzaro, T. Szumlak

Reach of heterogeneity

- ❑ In principle all devices from cell phones to large datacenters feature heterogeneous architectures
- ❑ Even an inexpensive laptop today can combine up to three different compute units: APU, CPU, and GPU
- ❑ Formally, we define a heterogeneous architecture, as one that accommodates more than one kind of compute units
- ❑ These compute units can be ASICs, FPGA chips, GPU cards, etc…
- ❑ The tricky part is to provide suitable interfaces

CPU-only heterogeneity: ARM’s big.LITTLE

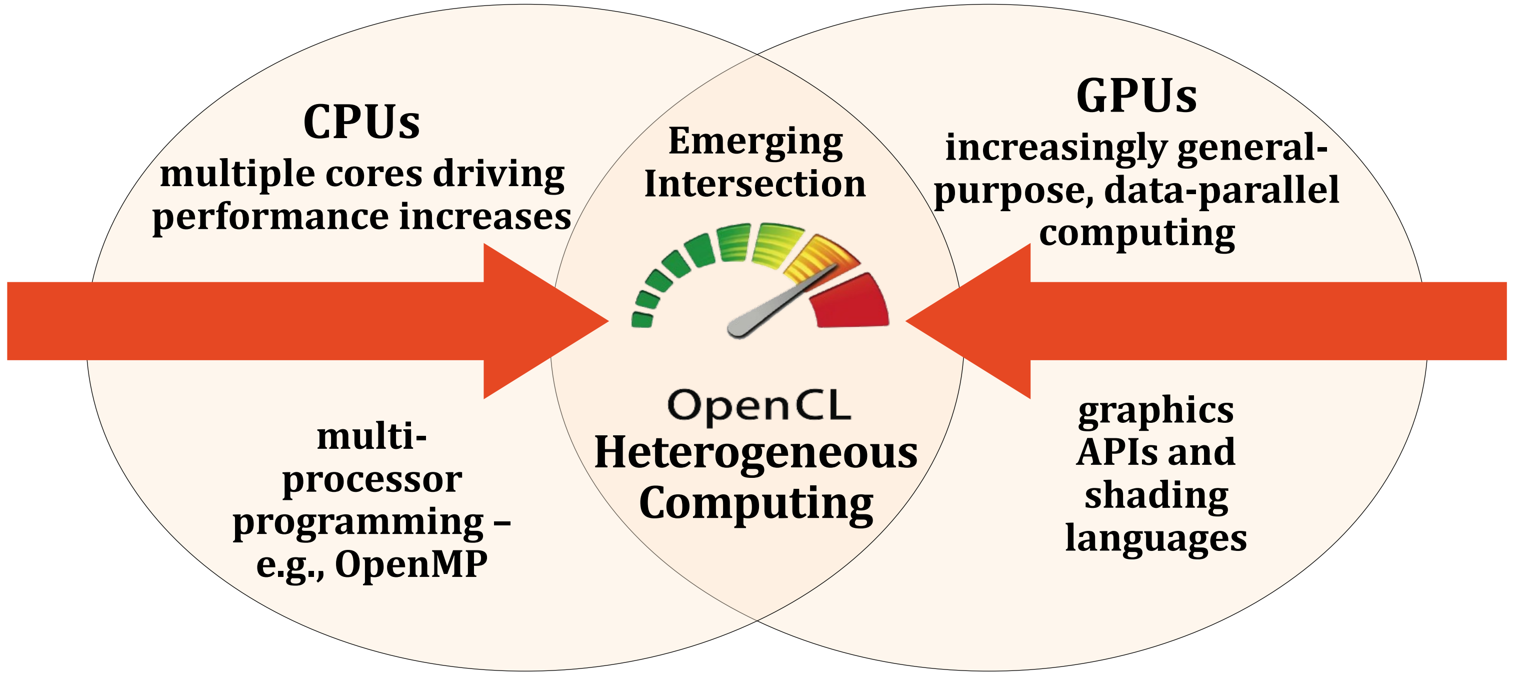
ARM’s big.LITTLE processing takes advantage of the variation smart devices require in performance by combining two very different processors together in a single SoC.


Linux Scheduler picks any **One Cluster** at a time,
But, no combinations
High Cluster is picked if at-least one High Core is needed, else **Low Cluster**
Either
Or
A57 **High Cluster** (High in Perf, Power) of 4 Cores
Cortex -A57
Cortex -A57
Cortex -A57
Cortex -A57
A53 **Low Cluster** (Low in Perf, Power) of 4 Cores
Cortex -A53
Cortex -A53
Cortex -A53
Cortex -A53

What is available in the market?

- **CUDA –** Compute Unified Device Architecture (the most popular proprietary platform for CPU-GPU systems).
- **OpenCL –** most popular open-source framework for executing code on heterogeneous architectures, very versatile, and very powerful.
- **OpenACC –** a programming standard to facilitate parallel computing applications.
- **OneAPI –** an open standard, trademarked by Intel, for a unified application programming interface intended to be used across different compute accelerator architectures, including GPUs, AI accelerators and FPGAs.

OpenCL: an open-source solution


CPUs
multiple cores driving performance increases
multi-processor programming – e.g., OpenMP
Emerging Intersection
OpenCL
Heterogeneous Computing
GPUs
increasingly general-purpose, data-parallel computing
graphics APIs and shading languages

Laying the foundation

- ❑ The fundamental goal is to **use all computation units** (resources) available on a given system.
- ❑ Exploits both data parallel (SIMD) and task parallel models.
- ❑ You create an OpenCL code by using an extension to C language.
- ❑ Providing abstraction of the underlying (architecture-level) parallelism.
- ❑ Different implementations (i.e., different libraries from AMD/ATI, NVIDIA, …) define platforms which in turn can enable the host system to interface with OpenCL-capable/OpenCL-enabled device (very similar to CUDA-enabled devices).
- ❑ OpenCL has its own particular structure.

Dissecting OpenCL

- Platform Layer API
  - Hardware abstraction layer
  - **Query** facility, **select,** and **initialize** compute devices (CD)
  - Create **compute contexts** and **task queues**
- Run-time API
  - **Execute** compute kernels
  - Scheduler to **manage the resources**: processing units and memory
- Language
  - C-based extension
  - A lot of goodies as built-in functions

Design goals

Use *all* computational resources the system.

Platform independence

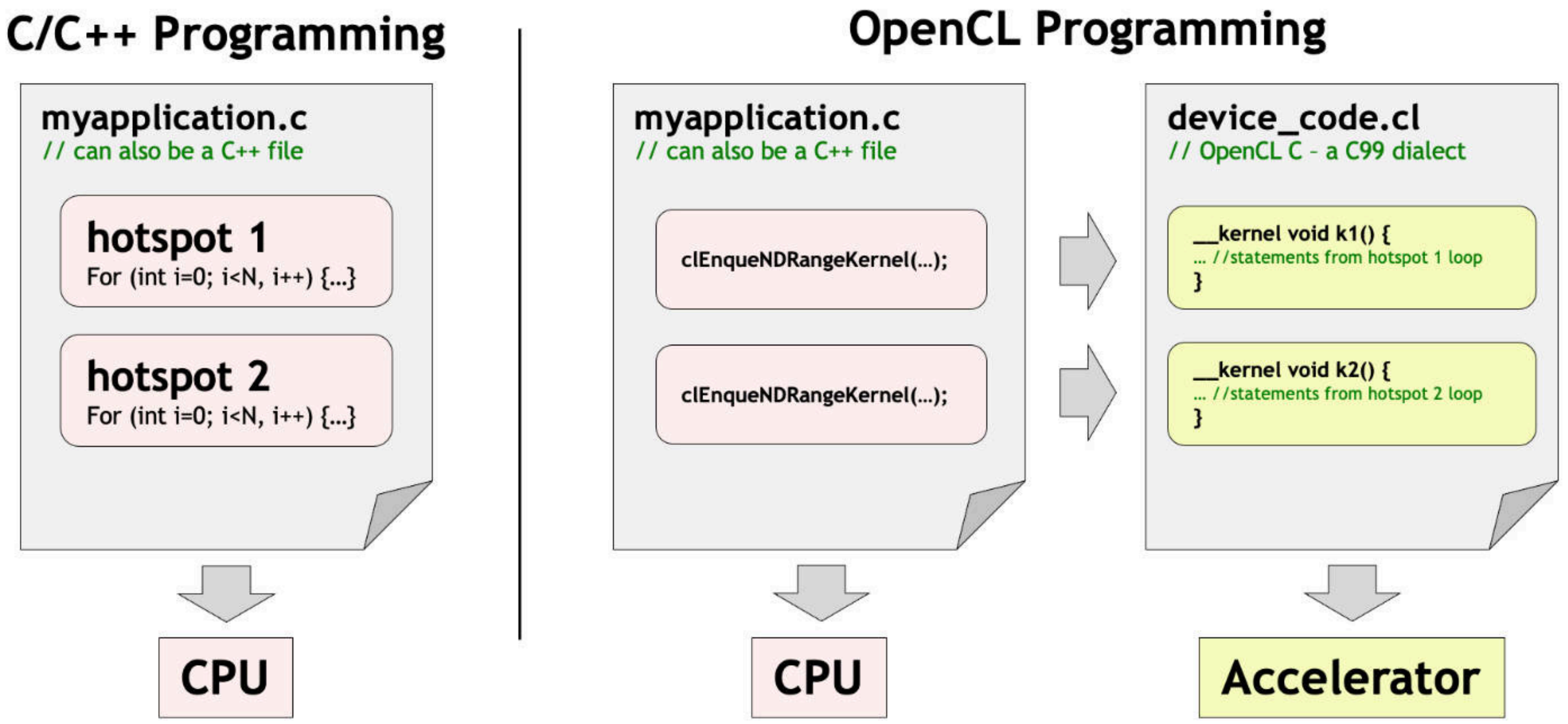
Provide *both* data and task parallel computational models.

Provide a programming model which *abstracts out* the specifics of the underlying hardware.

Specify accuracy of floating-point computations.

Support both desktop and handheld/portables.

OpenCL programming model



C/C++ Programming
myapplication.c
// can also be a C++ file
hotspot 1
For (int i=0; i<N, i++) {...}
hotspot 2
For (int i=0; i<N, i++) {...}
CPU
OpenCL Programming
myapplication.c
// can also be a C++ file
clEnqueNDRangeKernel(...);
clEnqueNDRangeKernel(...);
CPU
device_code.cl
// OpenCL C - a C99 dialect
__kernel void k1() {
... //statements from hotspot 1 loop
}
__kernel void k2() {
... //statements from hotspot 2 loop
}
Accelerator

High-level vision if similar to that of CUDA …

OpenCL platform model

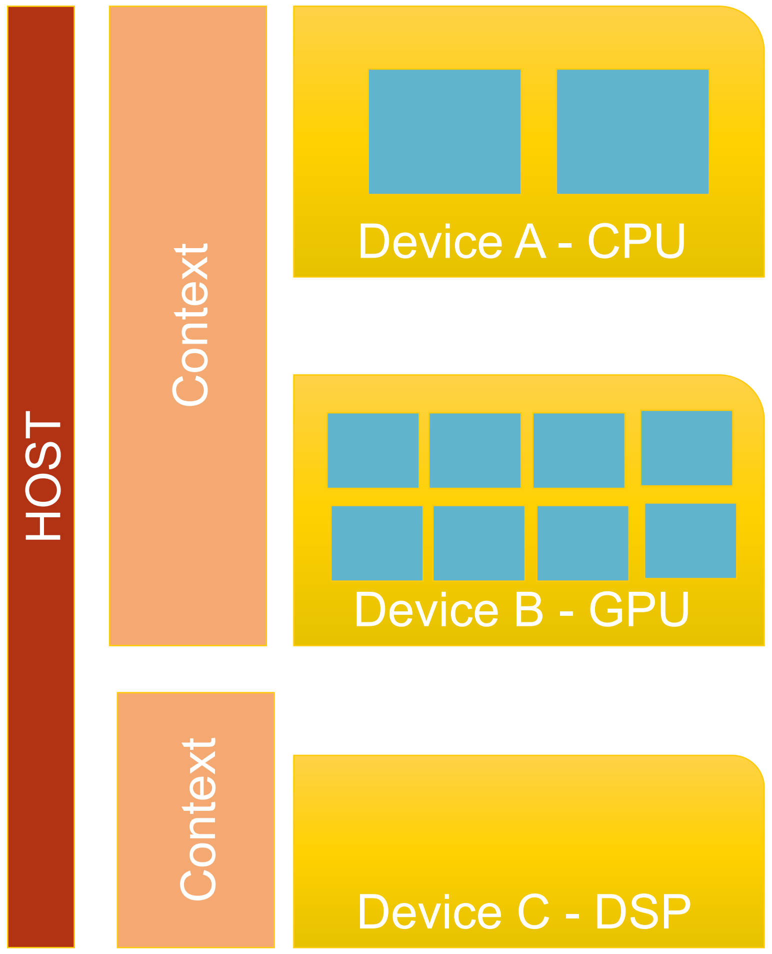
Host connected to one or more OpenCL devices

Device consists of one or more cores

Each device can be heterogeneous (e.g., APU)

Execution per processor may be SIMD or SPMD

Contexts group together devices and enable inter-device communication



OpenCL memory model

Private memory is available per work item

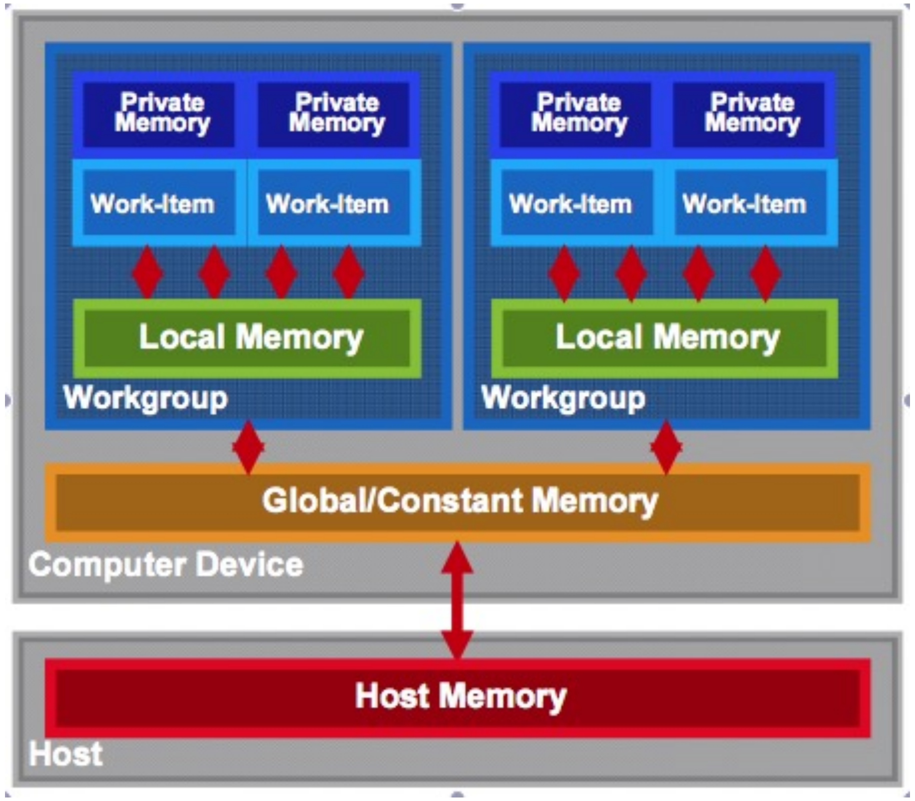
Local memory shared within workgroup

No synchronization between workgroups

Synchronization possible between work items in a workgroup

Global/constant memory for access by work-items – not synchronized

Host memory – access through the CPU



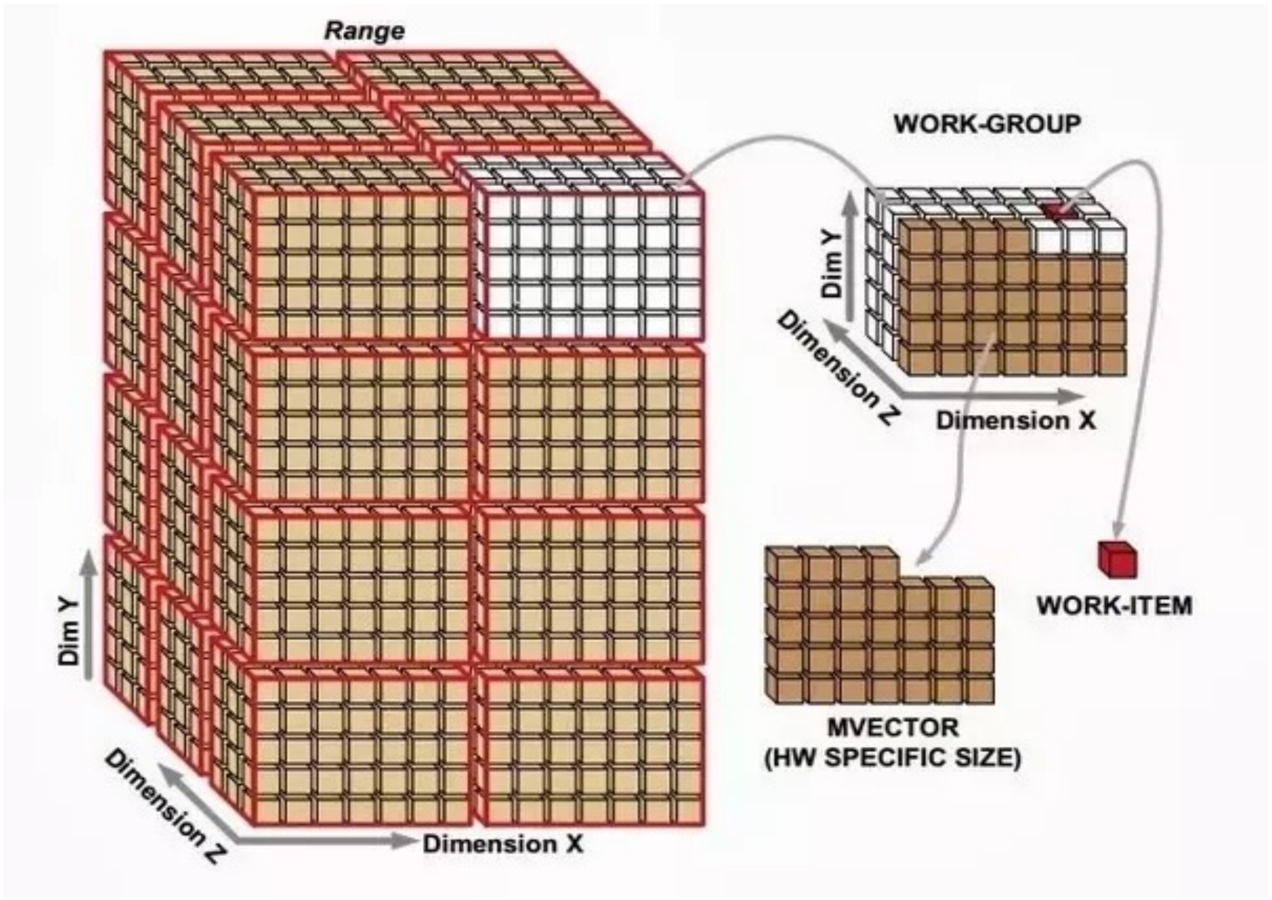
- Memory management is explicit
- Data should be moved from host->global->local and back

Work-groups and work-items

An OpenCL program is organized as a grid of **work-groups**. Each work-group is organized as a grid of **work-items**.

In terms of hardware, a work-group runs on a Compute Unit (CU) and a work-item runs on a Processing Element (PE).

One thread is assigned to each work-item.



Main OpenCL objects

Devices – multiple cores on CPU/GPU together taken as a single device.

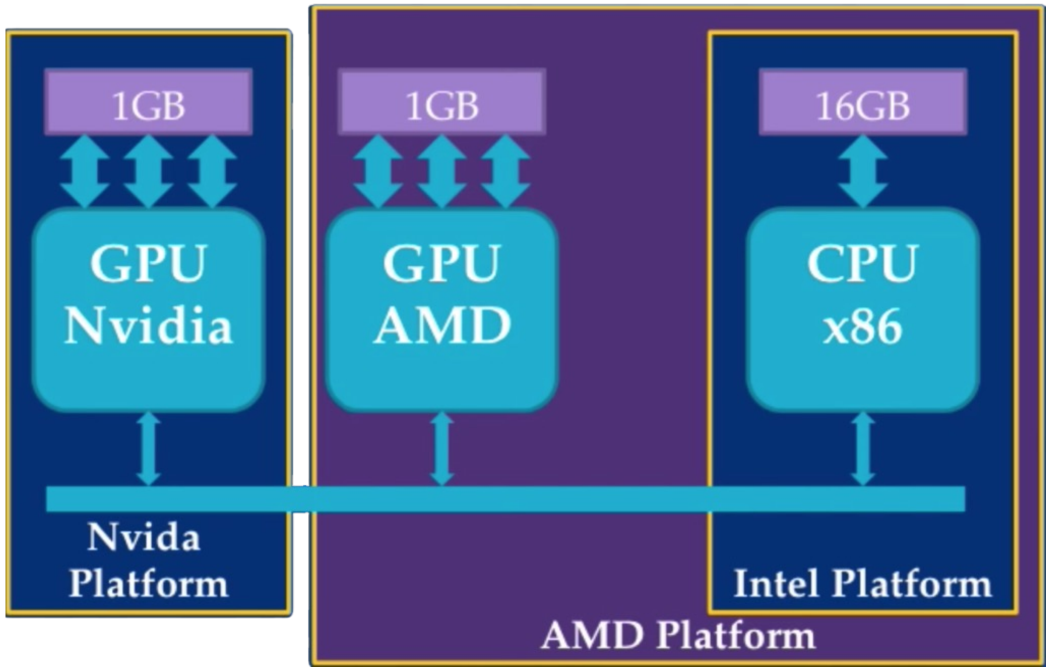
Kernels executed across all cores in a **data-parallel** fashion.

Contexts – Enable sharing between different devices

- Devices must be within the **same context** to be able to share

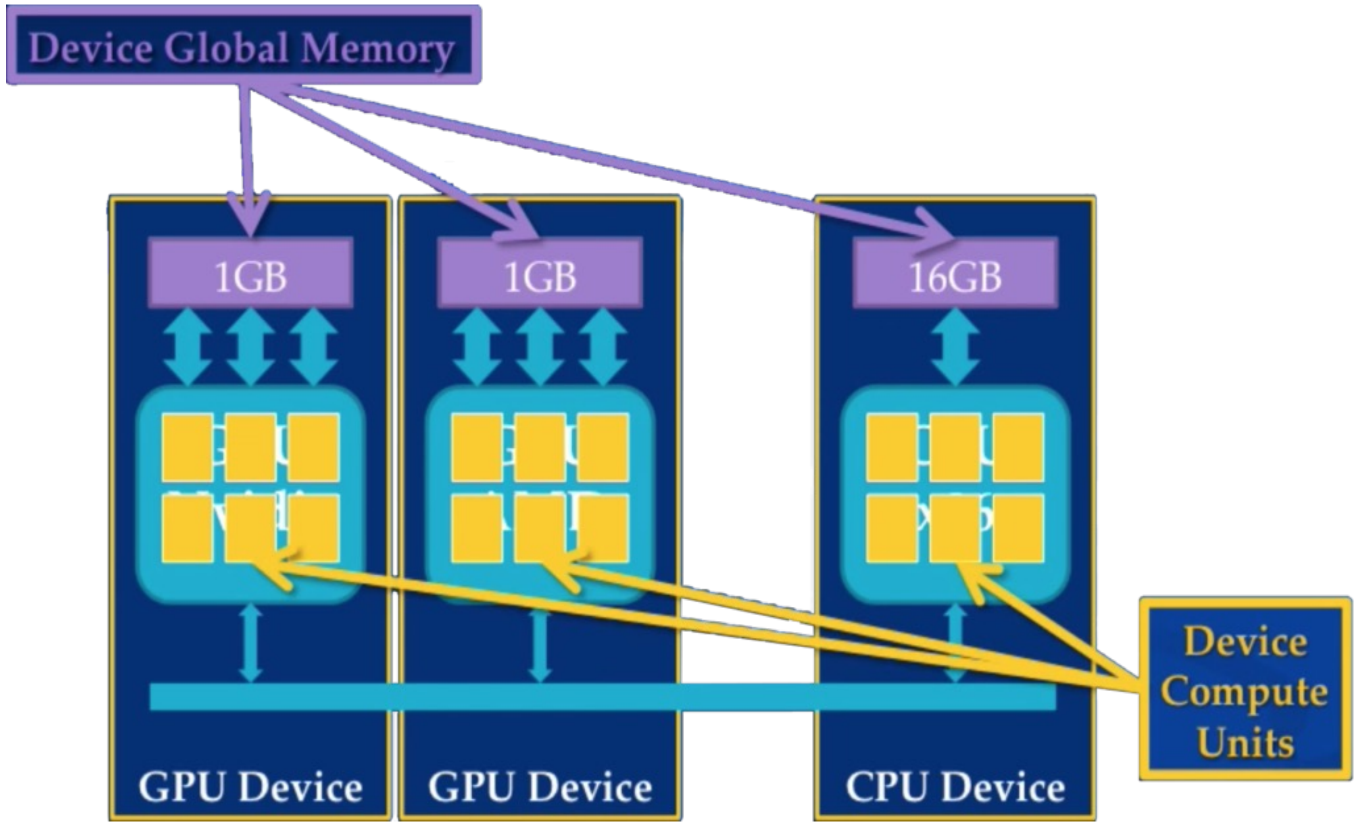
Queues – used for submitting work, **one per device**

Platforms



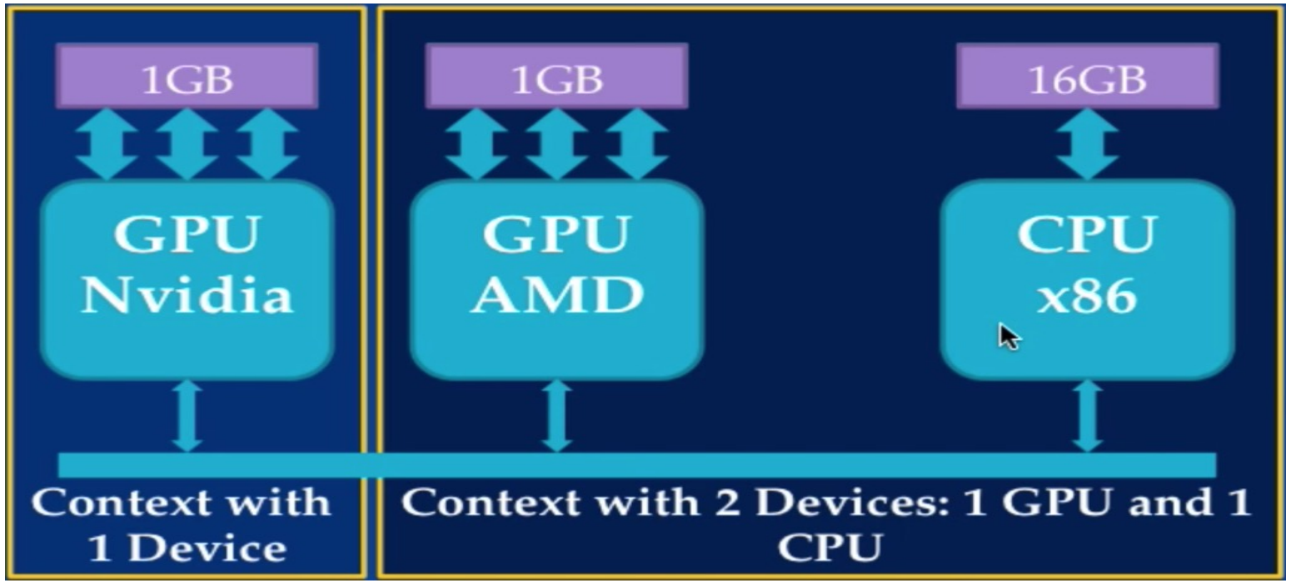
A platform is a *collection of devices*. The platform determines how data can be *shared* efficiently. If the platform supports *both* CPU and GPU, the vendor would have optimized data flow between the two devices. Data sharing within a platform is “more efficient” than across platforms.

Devices



The individual CPU/GPU are called devices. The CPU device can be shared as *both* an OpenCL device and the host processor. Devices (CPU/GPU) are connected via a bus. Each device has a memory attached to it limited by its peak bandwidth (arrows).

Contexts



The context is the **environment** within which the kernels execute. This environment includes **a set of devices.** All devices in a context must be in the *same platform*. The memory accessible to those devices. One or more command-queues used to schedule execution of a kernel(s) or operations on memory objects.

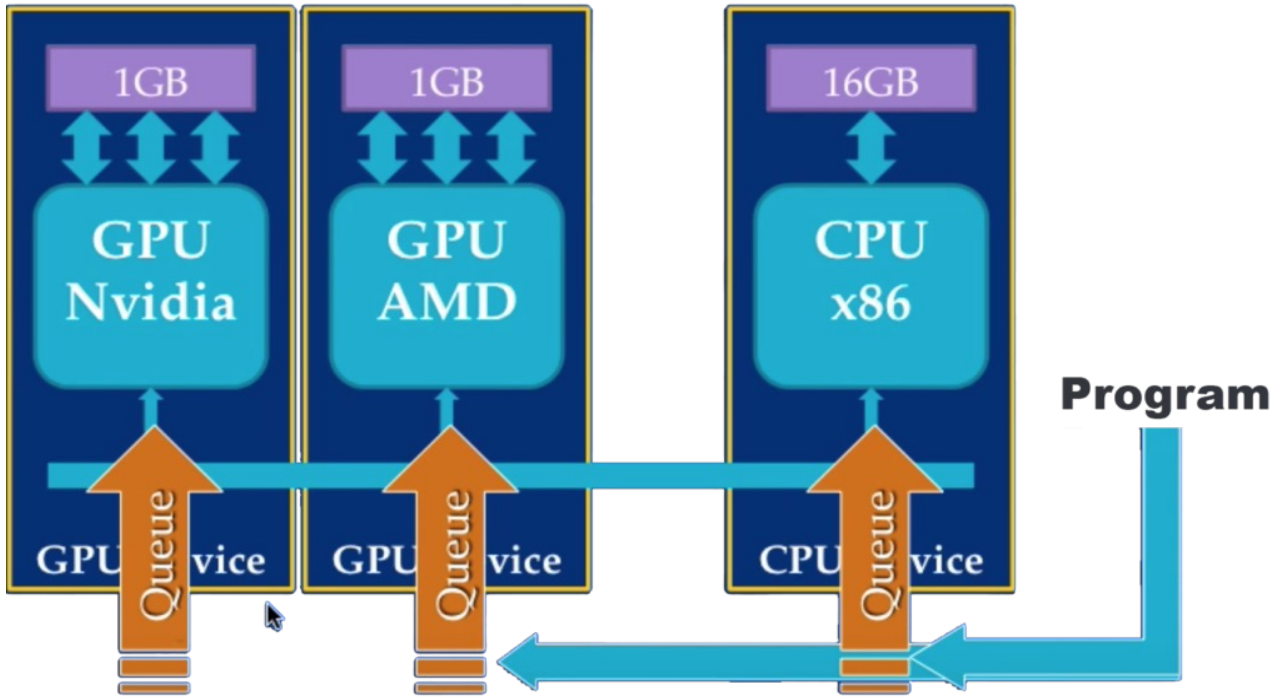
Contexts are used to contain and manage the “state of the world” in OpenCL. This includes:

Kernel execution commands

Memory commands – transfer or mapping of memory object data

Synchronization commands – constrains the order of commands

Command queues



To submit work to a device a command queue has to be created. The program can put work into this queue and eventually will make it to the top of the queue and get executed on the device. To execute on another device a new command queue has to be created. Thus, ***a command queue is needed for every device.*** This means there is *no* automatic distribution of work across devices.

Each command-queue points to a single device within a context. A single device can simultaneously be attached to multiple command queues. Both in-order and out-of-order queues are possible.

Data movement

No automatic data movement. The user gets full control of performance (and responsibility!) and must explicitly:

- Allocate global data.
- Write to it from the host.
- Allocate local data.
- Copy data from global to local (and back).

OpenCL kernel objects

Declared with a **kernel** *qualifier.*

Encapsulate a kernel function (pretty much like device functions in CUDA).

Kernel objects are created after the executable is built.

Execution:

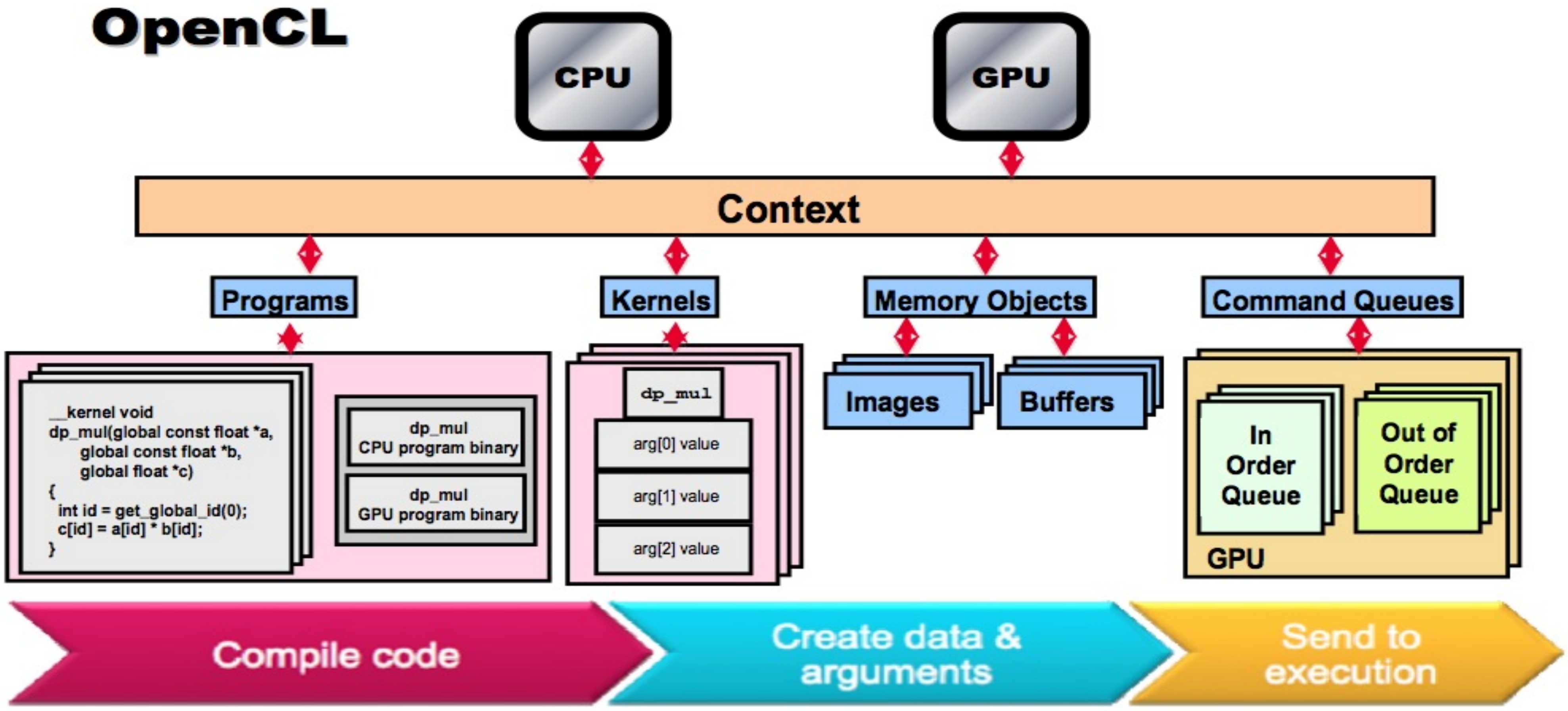
- Set the kernel arguments.
- Enqueue the kernel in a device queue.

Kernels are executed *asynchronously.*

Events used to track the execution status:

- Used for synchronizing execution of two kernels.
- clWaitForEvents(), clEnqueueMarker() etc.

Overall pipeline



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

Compilation, preparation, execution

1. **Setup**
    1. Get the devices (and platform)
    2. Create a context (for sharing between devices)
    3. Create command queues (for submitting work)
2. **Compilation**
    1. Create a program
    2. Build the program (compile)
    3. Create kernels
3. **Create memory objects**
4. **Enqueue writes to copy data to the GPU**
5. **Set the kernel arguments**
6. **Enqueue kernel executions**
7. **Enqueue reads to copy data back from the GPU**
8. **Wait for your commands to finish**
9. **Clean up OpenCL resources**

OpenCL is asynchronous. When we enqueue a command we have no idea when it will finish. By explicitly waiting, we make sure it is finished before continuing.

clGetDeviceIDs

Obtains the list of devices available on a platform.

cl_int **clGetDeviceIDs**(	cl_platform_id *platform*,
	cl_device_type *device_type*,
	cl_uint *num_entries*,
	cl_device_id **devices*,
	cl_uint **num_devices*)

clGetDeviceIDs

Parameters
platform
Refers to the platform ID returned by clGetPlatformIDs or can be NULL. If *platform* is NULL, the behavior is implementation-defined.
device_type
A bitfield that identifies the type of OpenCL device. The *device_type* can be used to query specific OpenCL devices or all OpenCL devices available. The valid values for *device_type* are specified in the following table.
num_entries
The number of cl_device entries that can be added to *devices*. If *devices* is not NULL, the *num_entries* must be greater than zero.
devices
A list of OpenCL devices found. The cl_device_id values returned in *devices* can be used to identify a specific OpenCL device. If *devices* argument is NULL, this argument is ignored. The number of OpenCL devices returned is the mininum of the value specified by *num_entries* or the number of OpenCL devices whose type matches *device_type*.
num_devices
The number of OpenCL devices available that match *device_type*. If *num_devices* is NULL, this argument is ignored.

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.

Implementing the SAXPY routine in OpenCL

SAXPY can be called the "Hello World" of OpenCL.

In the simplest terms, the first OpenCL sample shall compute A = alpha*B + C, where alpha is a constant and A, B, and C are vectors of an arbitrary size n.

In *linear algebra* terms, this operation is called **SAXPY (Single precision real Alpha X plus Y)**.

Each multiplication and addition operation is independent of the other. So, this is a data parallel problem.

Implementing the SAXPY routine in OpenCL

A simple C program would look something like the following code:

```
void saxpy(int n, float a, float *b, float *c)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
```

Implementing the SAXPY routine in OpenCL

```
#include <stdio.h>

…

//OpenCL kernel which is run for every work item created.

const char *saxpy_kernel =

"__kernel                              \n"

"void saxpy_kernel(float alpha,     \n"

"               __global float *A,      \n"

"               __global float *B,      \n"

"               __global float *C)      \n"

"{                                     \n"

"    //Get the index of the work-item        \n"

"    int index = get_global_id(0);         \n"

"    C[index] = alpha* A[index] + B[index]; \n"

"}                                     \n";
```

An OpenCL code consists of the host code and the device code. This is the code which is compiled at run time and runs on the selected device. The following sample code computes A = alpha*B + C, where A, B, and C are vectors (arrays) of size given by the VECTOR_SIZE variable:

Implementing the SAXPY routine in OpenCL

```
int main(void) {
 int i;
 // Allocate space for vectors A, B and C
 float alpha = 2.0;
 float *A = (float*)malloc(sizeof(float)*VECTOR_SIZE);
 float *B = (float*)malloc(sizeof(float)*VECTOR_SIZE);
 float *C = (float*)malloc(sizeof(float)*VECTOR_SIZE);
 for(i = 0; i < VECTOR_SIZE; i++)
 {
  A[i] = i;
  B[i] = VECTOR_SIZE - i;
  C[i] = 0;
 }
```

Implementing the SAXPY routine in OpenCL

```
// Get platform and device information
  cl_platform_id * platforms = NULL;
  cl_uint     num_platforms;
  //Set up the Platform
  cl_int clStatus = clGetPlatformIDs(0, NULL, &num_platforms);
  platforms = (cl_platform_id *)
  malloc(sizeof(cl_platform_id)*num_platforms);
  clStatus = clGetPlatformIDs(num_platforms, platforms, NULL);


  //Get the devices list and choose the device you want to run on
  cl_device_id     *device_list = NULL;
  cl_uint          num_devices;
  clStatus = clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_GPU, 0,NULL,
     &num_devices);
```

Implementing the SAXPY routine in OpenCL

```
device_list = (cl_device_id *)
  malloc(sizeof(cl_device_id)*num_devices);
  clStatus = clGetDeviceIDs( platforms[0],CL_DEVICE_TYPE_GPU, num_devices,
     device_list, NULL);

  // Create one OpenCL context for each device in the platform
  cl_context context;
  context = clCreateContext( NULL, num_devices, device_list, NULL, NULL, &clStatus);

  // Create a command queue
  cl_command_queue command_queue = clCreateCommandQueue(context,
     device_list[0], 0, &clStatus);
```

Implementing the SAXPY routine in OpenCL

```
// Create memory buffers on the device for each vector

cl_mem A_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,VECTOR_SIZE *
    sizeof(float), NULL, &clStatus);

cl_mem B_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,VECTOR_SIZE *
    sizeof(float), NULL, &clStatus);

cl_mem C_clmem = clCreateBuffer(context, CL_MEM_WRITE_ONLY,VECTOR_SIZE *
    sizeof(float), NULL, &clStatus);


// Copy the Buffer A and B to the device

clStatus = clEnqueueWriteBuffer(command_queue, A_clmem, CL_TRUE, 0,
    VECTOR_SIZE * sizeof(float), A, 0, NULL, NULL);

clStatus = clEnqueueWriteBuffer(command_queue, B_clmem, CL_TRUE, 0,
    VECTOR_SIZE * sizeof(float), B, 0, NULL, NULL);
```

Implementing the SAXPY routine in OpenCL

```
// Create a program from the kernel source. (interesting really!)

cl_program program = clCreateProgramWithSource(context, 1,(const char
   **)&saxpy_kernel, NULL, &clStatus);


// Build the program

clStatus = clBuildProgram(program, 1, device_list, NULL, NULL, NULL);


// Create the OpenCL kernel

cl_kernel kernel = clCreateKernel(program, "saxpy_kernel", &clStatus);

// Set the arguments of the kernel

clStatus = clSetKernelArg(kernel, 0, sizeof(float), (void *)&alpha);

clStatus = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&A_clmem);

clStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&B_clmem);

clStatus = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *)&C_clmem);
```

Implementing the SAXPY routine in OpenCL

```
// Execute the OpenCL kernel on the list

size_t global_size = VECTOR_SIZE; // Process the entire lists

size_t local_size = 64;          // Process one item at a time

clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
   &global_size, &local_size, 0, NULL, NULL);


// Read the cl memory C_clmem on device to the host variable C

clStatus = clEnqueueReadBuffer(command_queue, C_clmem, CL_TRUE, 0,
   VECTOR_SIZE * sizeof(float), C, 0, NULL, NULL);


// Clean up and wait for all the comands to complete.

clStatus = clFlush(command_queue);

clStatus = clFinish(command_queue);
```

Implementing the SAXPY routine in OpenCL

```
// Display the result to the screen
for(i = 0; i < VECTOR_SIZE; i++)
  printf("%f * %f + %f = %f\n", alpha, A[i], B[i], C[i]);
// Finally release all OpenCL allocated objects and host buffers.
clStatus = clReleaseKernel(kernel);
clStatus = clReleaseProgram(program);
clStatus = clReleaseMemObject(A_clmem);
clStatus = clReleaseMemObject(B_clmem);
clStatus = clReleaseMemObject(C_clmem);
clStatus = clReleaseCommandQueue(command_queue);
clStatus = clReleaseContext(context);
free(A); free(B); free(C);
free(platforms); free(device_list);
return 0;}
```

Running the code on a different device

To make OpenCL run the kernel on the CPU, you can change the
enum CL_DEVICE_TYPE_GPU to CL_DEVICE_TYPE_CPU in the call to clGetDeviceIDs.

This shows how easy it is to make an OpenCL program run on different compute devices.

If you are running a multi-GPU hardware system, then you will have to modify the code to use the appropriate device ID.

Summary

OpenCL is portable and high-performance framework:

- Targets computationally intensive algorithms.
- Tries to utilize all computational resources.
- Accommodates well defined memory and computational model.

An efficient parallel programming language:

- C99 with extensions for enabling both task and data parallelism.
- Set of built in functions for synchronization, math and memory operations.

Open standard for parallel computing across heterogeneous collection of devices.

CUDA vs OpenCL: terminology

Thread	Work-item
Thread-block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory
__global__ function	__kernel function
__device__ function	no qualification needed
__constant__ variable	__constant variable
__device__ variable	__global variable
__shared__ variable	__local variable

CUDA vs OpenCL: performance

OpenCL assures a portable language for GPU programming, which is adept at targeting very unrelated parallel processing devices. This in no way means that a code is guaranteed to run on all devices if at all due to the fact that most have very different feature sets. Some extra effort has to be put in to make the code run on multiple devices while avoiding vendor-specific extension.

Unlike the CUDA kernel, an OpenCL kernel can be compiled at *runtime*, which would add up to an OpenCL's running time. However, on the other hand, this just-in-time compile could allow the compiler to generate code that will make better use of the target GPU.

CUDA, is developed by the same company that develops the hardware on which it executes its functions, which is why one may expect it to better match the computing characteristics of the GPU, and therefore offering more access to features and better performance.

However, performance wise, the compiler (and ultimately the programmer) is what makes each interface faster as both can fully utilize hardware. The performance will be dependent on some variables, including code quality, algorithm type and hardware type.

CUDA vs OpenCL: vendor support

There is only one vendor for **CUDA** implementation and that is its proprietor, NVIDIA.

OpenCL, however, has been implemented by a vast array of vendors including but not limited to:

- AMD: Intel and AMD chips and GPU's are supported.
- Radeon 5xxx, 6xxx, 7xxx series, R9xxx series are supported
- All CPUs support OpenCL 1.2 only
- NVIDIA: NVIDIA GeForce 8600M GT, GeForce 8800 GT, GeForce 8800 GTS, GeForce 9400M, GeForce 9600M GT, GeForce GT 120, GeForce GT 130, ATI Radeon 4850, Radeon 4870, and likely more are supported.
- Apple (MacOS X only is supported)
- Host CPUs as compute devices are supported
- CPU, GPU, and "MIC" (Xeon Phi).

CUDA vs OpenCL: portability

This is likely the most recognized difference between the two as **CUDA** runs on only NVIDIA GPUs while **OpenCL** is an open industry standard and runs on NVIDIA, AMD, Intel, and other hardware devices.

Also, OpenCL provides for CPU fallback and as such code maintenance is easier while on the other hand CUDA does not provide CPU fallback, which makes developers put if-statements in their codes that help to distinguish between the presence of a GPU device at runtime or its absence.

CUDA vs OpenCL: commercial vs open-source

Another highly recognized difference between **CUDA** and **OpenCL** is that OpenCL is open-source and CUDA is a proprietary framework of NVIDIA. This difference brings its own pros and cons and the general decision on this has to do with your app of choice.

Generally, if the app of your choice supports both CUDA and OpenCL, going with CUDA is the best option as it generates better performance results in this scenario. This is because NVIDIA provides top notch support. If some apps are CUDA based and others have OpenCL support, a recent NVIDIA card will help you get the most out of CUDA enabled apps while having good compatibility in non-CUDA apps.

However, if all your apps of choice are OpenCL supported then the decision is already made for you.

CUDA vs OpenCL: OS support

CUDA is able to run on Windows, Linux, and MacOS, but *only* using NVIDIA hardware. However, **OpenCL** is available to run on almost any operating system and most hardware varieties. When it comes to the OS support comparison the chief deciding factor still remains the hardware as CUDA is able to run on the leading operating systems while OpenCL runs on almost all.

The hardware distinction is what really sets the comparison. With CUDA having a requirement of only the use of NVIDIA hardware, while with OpenCL the hardware is so not specified. This distinction has its own pros and cons.

CUDA vs OpenCL: libraries

Libraries are key to GPU Computing, because they give access to a set of functions which have already been finetuned to take advantage of data-parallelism. **CUDA** comes in very strong in this category as it has support for templates and free raw math libraries which embody high performance math routines:

- cuBLAS – Complete BLAS Library
- cuRAND – Random Number Generation (RNG) Library
- cuSPARSE – Sparse Matrix Library
- NPP – Performance Primitives for Image & Video Processing
- cuFFT – Fast Fourier Transforms Library
- Thrust – Templated Parallel Algorithms & Data Structures
- h – C99 floating-point Library

OpenCL has alternatives which can be easily built and have matured in recent times, however nothing like the CUDA libraries. An example of which is the ViennaCL. AMD's OpenCL libraries also have an added bonus of not only running on AMD devices but additionally on all OpenCL compliant devices.

CUDA vs OpenCL: programming

CUDA allows for developers to write their software in C or C++ because it is only a platform and programming model not a language or API. Parallelization is achieved by the employment of CUDA keywords.

On the other hand, **OpenCL** does not permit for writing code in C++; however, it provides an environment resembling the C programming language for work and permits for work with GPU resources directly.