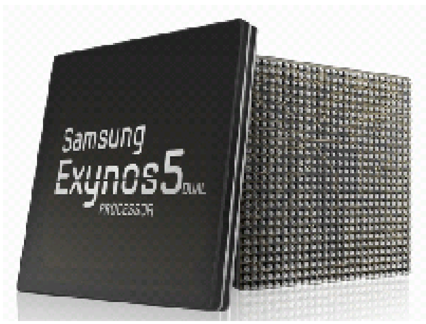
AN OVERVIEW OF OPENCL

It’s a Heterogeneous world

A modern computing platform includes:

- One or more CPUs
- One of more GPUs
- DSP processors
- Accelerators
- … other?



E.g. Samsung® Exynos 5:

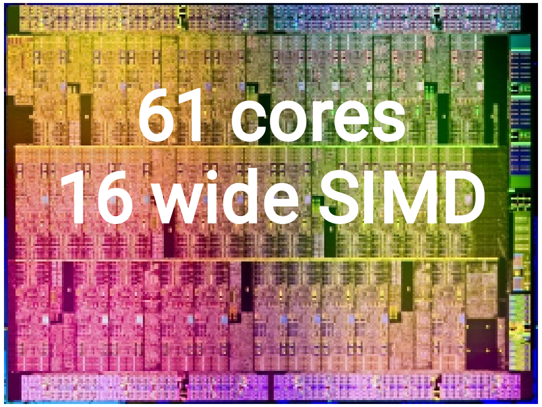
- Dual core ARM A15 1.7GHz, Mali T604 GPU

E.g. Intel XXX with IRIS

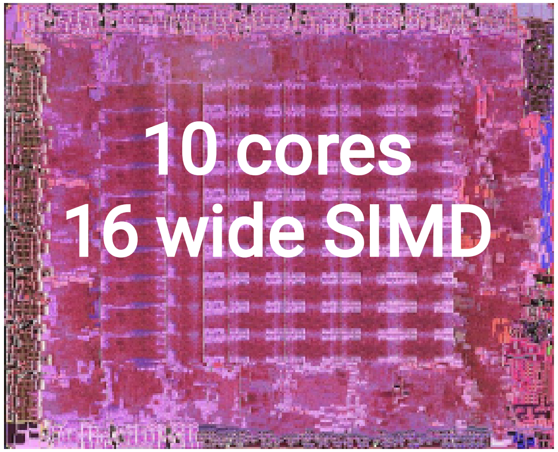
OpenCL lets Programmers write a single <u>portable</u> program that uses <u>ALL</u> resources in the heterogeneous platform

Microprocessor trends

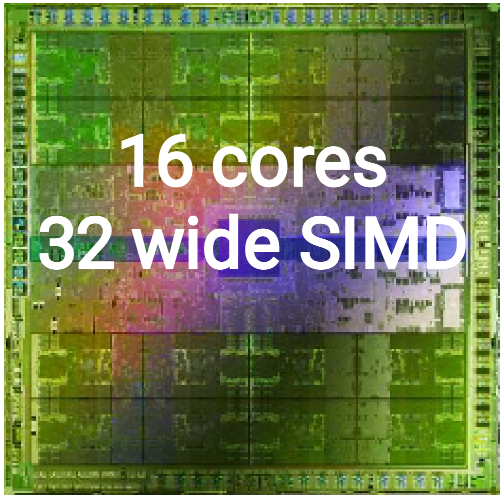
Individual processors have many (possibly heterogeneous) cores.



Intel® Xeon Phi™ coprocessor



ATI™ RV770

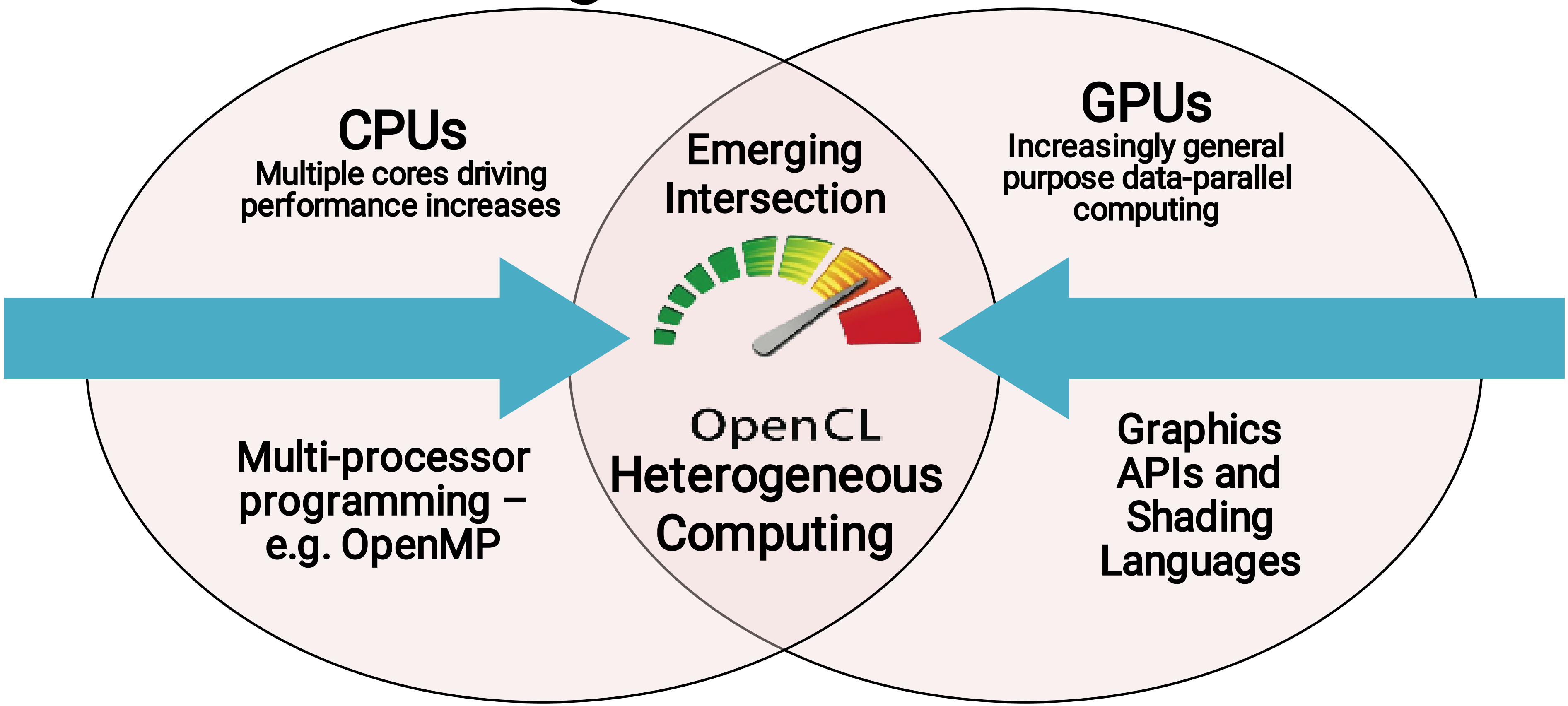


NVIDIA® Tesla® C2090

The Heterogeneous many-core challenge:

How are we to build a software ecosystem for the Heterogeneous many core platform?

Industry Standards for Programming Heterogeneous Platforms



OpenCL – Open Computing Language

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

The origins of OpenCL

AMD

ATI

Merged, needed commonality across products

NVIDIA – GPU vendor – wants to steal market share from CPU

Intel – CPU vendor – wants to steal market share from GPU

Apple – Was tired of recoding for many core, GPUs. Pushed vendors to standardize.

Wrote a rough draft straw man API

Khronos Compute group formed

- ARM
- Nokia
- IBM
- Sony
- Qualcomm
- Imagination
- TI
- + many more

OpenCL

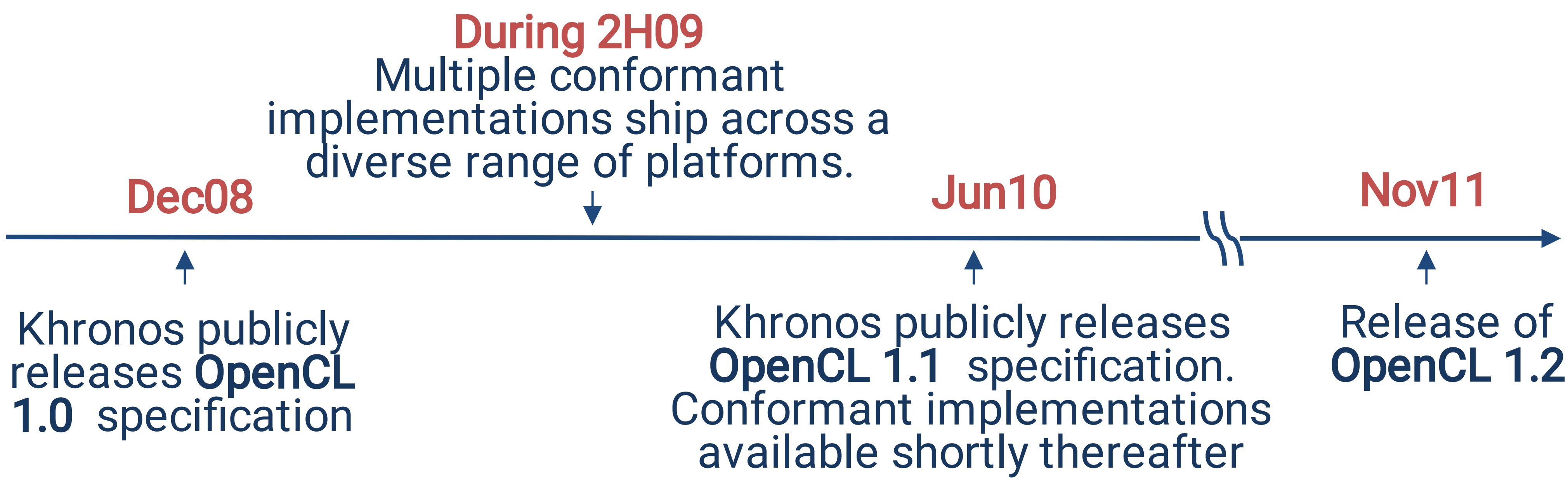
Third party names are the property of their owners.

OpenCL Working Group within Khronos

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers.
- OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.

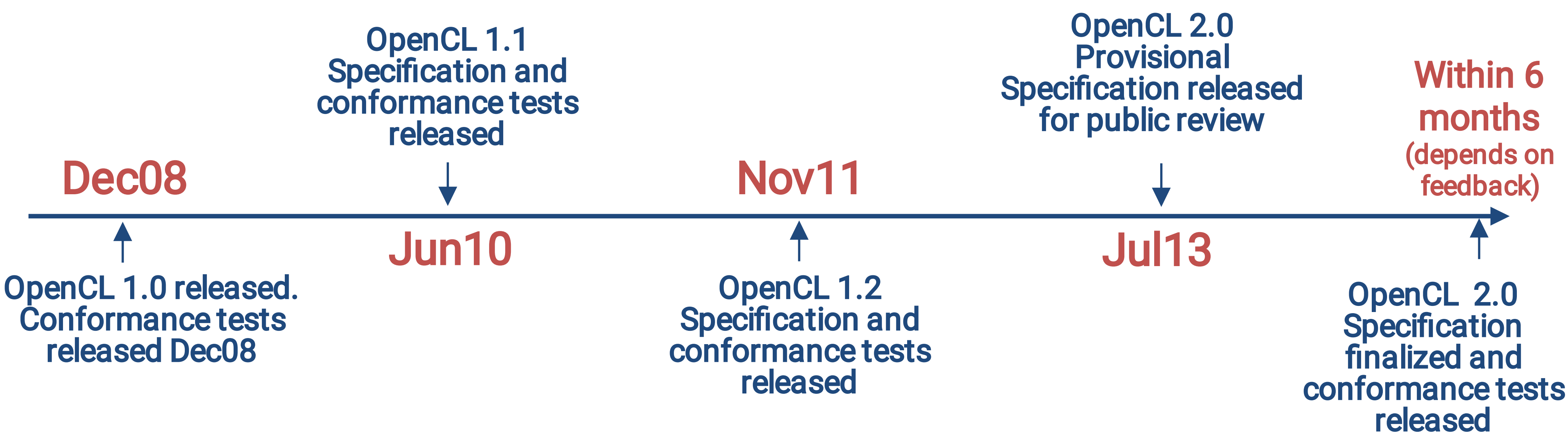
OpenCL Timeline

- Launched Jun'08 … 6 months from "strawman" to OpenCL 1.0
- Rapid innovation to match pace of hardware innovation
  - 18 months from 1.0 to 1.1 and from 1.1 to 1.2
  - Goal: a new OpenCL every 18-24 months
  - Committed to backwards compatibility to protect software investments



OpenCL Timeline

- Launched Jun'08 … 6 months from "strawman" to OpenCL 1.0
- Rapid innovation to match pace of hardware innovation
  - 18 months from 1.0 to 1.1 and from 1.1 to 1.2
  - Goal: a new OpenCL every 18-24 months
  - Committed to backwards compatibility to protect software investments

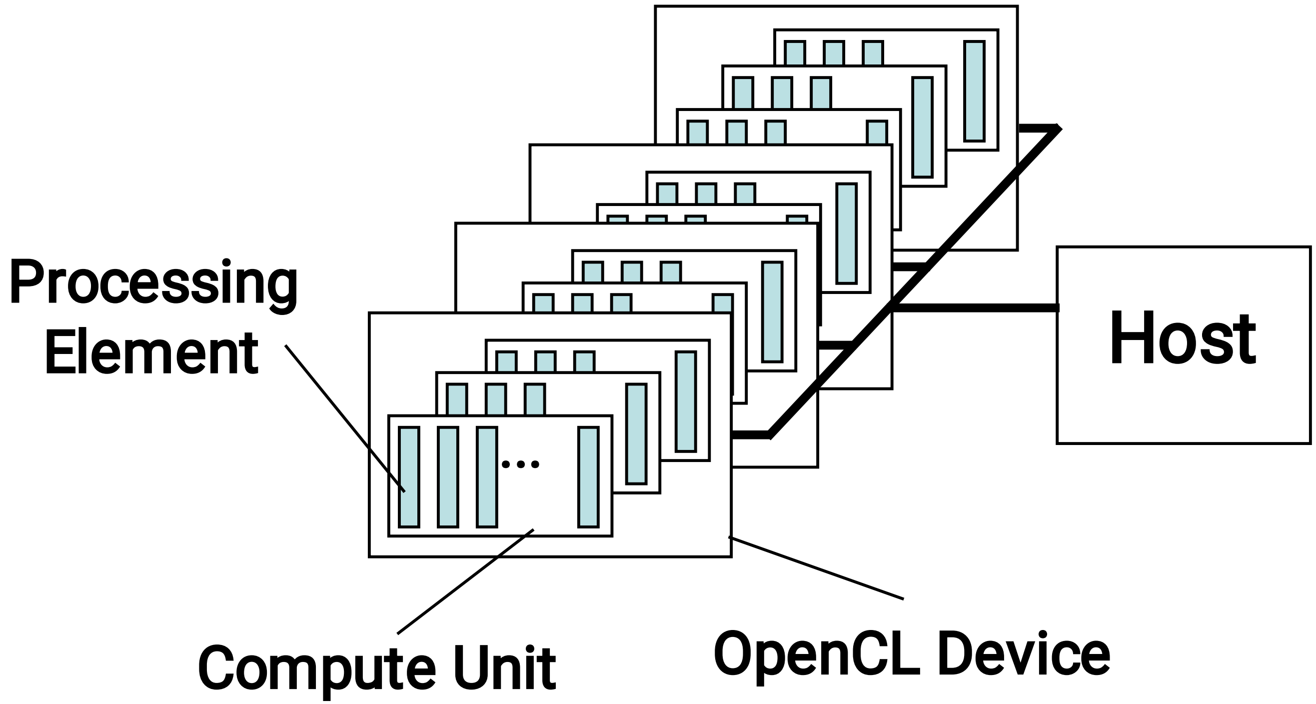


OpenCL: From cell phone to supercomputer

- OpenCL Embedded profile for mobile and embedded silicon
  - Relaxes some data type and precision requirements
  - Avoids the need for a separate "ES" specification
- Khronos APIs provide computing support for imaging & graphics
  - Enabling advanced applications in, e.g., Augmented Reality
- OpenCL will enable parallel computing in new markets
  - Mobile phones, cars, avionics

A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet

OpenCL Platform Model



- One *Host* and one or more *OpenCL Devices*
  - Each OpenCL Device is composed of one or more *Compute Units*
    - Each Compute Unit is divided into one or more *Processing Elements*
- Memory divided into *host memory* and *device memory*

OpenCL Platform Example (One node, two CPU sockets, two GPUs)

CPUs:

- Treated as one OpenCL device
  - One CU per core
  - 1 PE per CU, or if PEs mapped to SIMD lanes, *n* PEs per CU, where *n* matches the SIMD width
- Remember:
  - the CPU will also have to be its own host!

GPUs:

- Each GPU is a separate OpenCL device
- Can use CPU and all GPU devices concurrently through OpenCL

CU = Compute Unit; PE = Processing Element

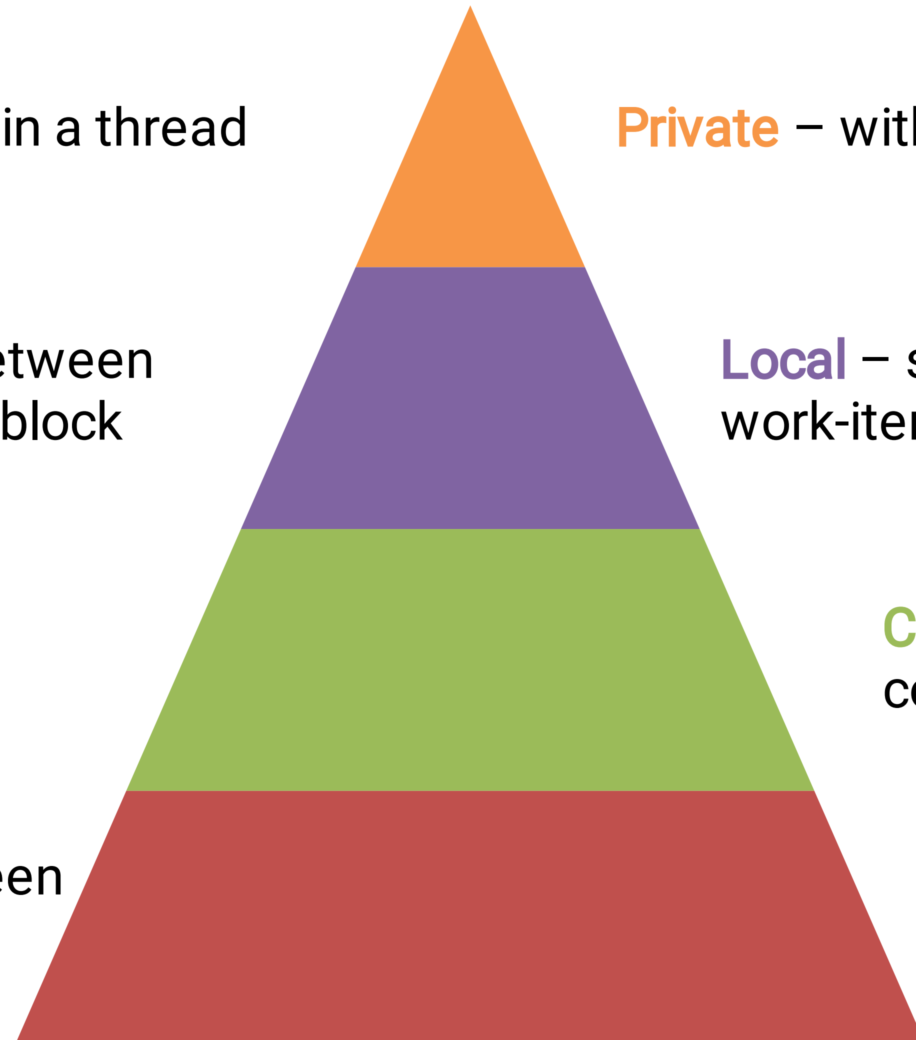
RELATING CUDA TO OPENCL

Introduction to OpenCL

- If you have CUDA code, you’ve already done the hard work!
  - I.e. working out how to split up the problem to run effectively on a many-core device

- Switching between CUDA and OpenCL is mainly changing the host code syntax
  - Apart from indexing and naming conventions in the kernel code (simple to change!)

Memory Hierarchy Terminology



CUDA	OpenCL
Local – within a thread	**Private** – within a work-item
Shared – shared between threads in a thread block	**Local** – shared between work-items in a work-group
Constant – a cache for constant memory	**Constant** – a cache for constant memory
Device – shared between all thread blocks	**Global** – shared between all work-groups

Allocating and copying memory

	CUDA C	OpenCL C
Allocate	float* d_x; cudaMalloc(&d_x, sizeof(float)*size);	cl_mem d_x = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*size, NULL, NULL);
Host to Device	cudaMemcpy(d_x, h_x, sizeof(float)*size, cudaMemcpyHostToDevice);	clEnqueueWriteBuffer(queue, d_x, CL_TRUE, 0, sizeof(float)*size, h_x, 0, NULL, NULL);
Device to Host	cudaMemcpy(h_x, d_x, sizeof(float)*size, cudaMemcpyDeviceToHost);	clEnqueueReadBuffer(queue, d_x, CL_TRUE, 0, sizeof(float)*size, h_x, 0, NULL, NULL);

Allocating and copying memory

	CUDA C	OpenCL C++

Allocate	`float* d_x;`
	`cudaMalloc(&d_x,`
	`   sizeof(float)*size);`
		`cl::Buffer`
		`  d_x(begin(h_x), end(h_x), true);`

Host to Device	`cudaMemcpy(d_x, h_x,`
	`   sizeof(float)*size,`
	`   cudaMemcpyHostToDevice);`
		`cl::copy(begin(h_x), end(h_x),`
		`      d_x);`

Device to Host	`cudaMemcpy(h_x, d_x,`
	`   sizeof(float)*size,`
	`   cudaMemcpyDeviceToHost);`
		`cl::copy(d_x,`
		`      begin(h_x), end(h_x));`

Declaring dynamic local/shared memory

CUDA C

1. Define an array in the kernel source as extern

```
__shared__ int array[];
```

2. When executing the kernel, specify the third parameter as size in bytes of shared memory

```
func<<<num_blocks,
 num_threads_per_block,
 shared_mem_size>>>(args);
```

OpenCL C

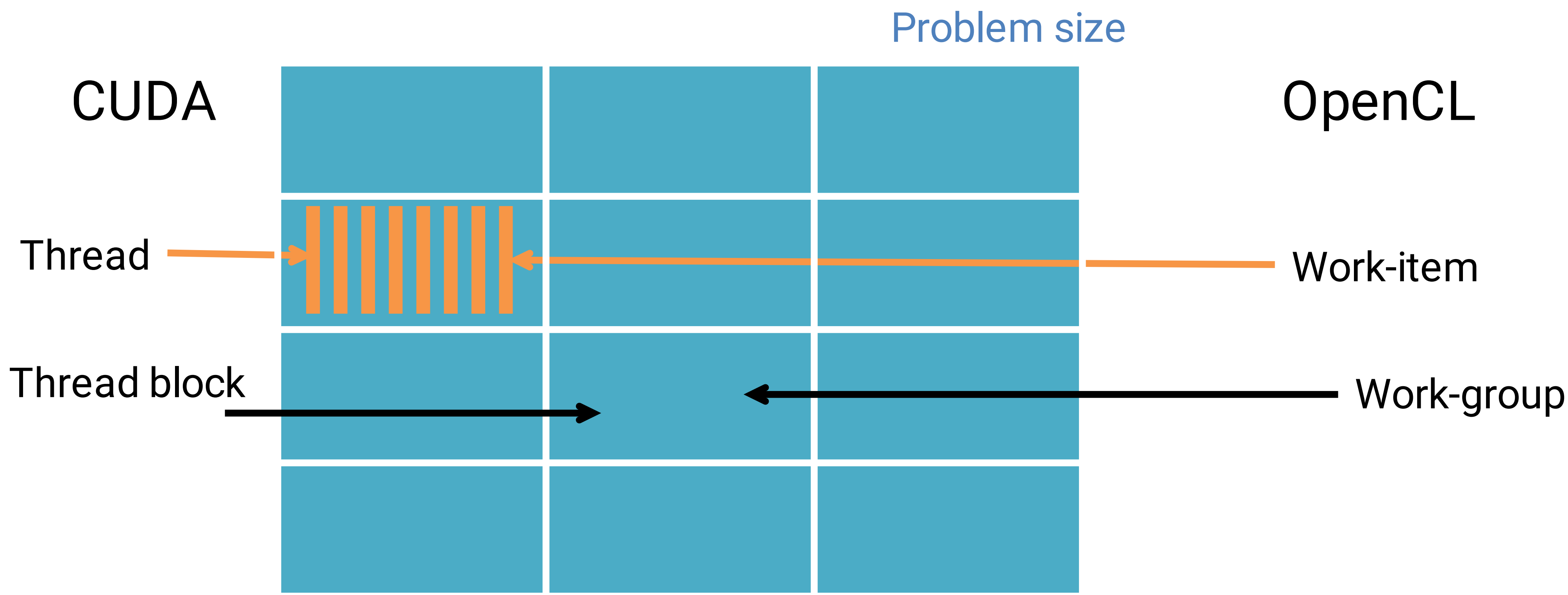
1. Have the kernel accept a local array as an argument

```
__kernel void func(
    __local int *array) {}
```

2. Specify the size by setting the kernel argument

```
clSetKernelArg(kernel, 0,
 sizeof(int)*num_elements,
 NULL);
```

Dividing up the work



- To enqueue the kernel
  - CUDA – specify the number of thread blocks and threads per block
  - OpenCL – specify the problem size and (optionally) number of work-items per work-group

Enqueue a kernel (C)

CUDA C

```
dim3 threads_per_block(30,20);

dim3 num_blocks(10,10);

kernel<<<num_blocks,
        threads_per_block>>>();
```

OpenCL C

```
const size_t global[2] =
            {300, 200};

const size_t local[2] =
            {30, 20};

clEnqueueNDRangeKernel(
     queue, &kernel,
     2, 0, &global, &local,
     0, NULL, NULL);
```

Enqueue a kernel (C++)

CUDA C

```
dim3 threads_per_block(30,20);

dim3 num_blocks(10,10);

kernel<<<num_blocks,
  threads_per_block>>>(…);
```

OpenCL C++

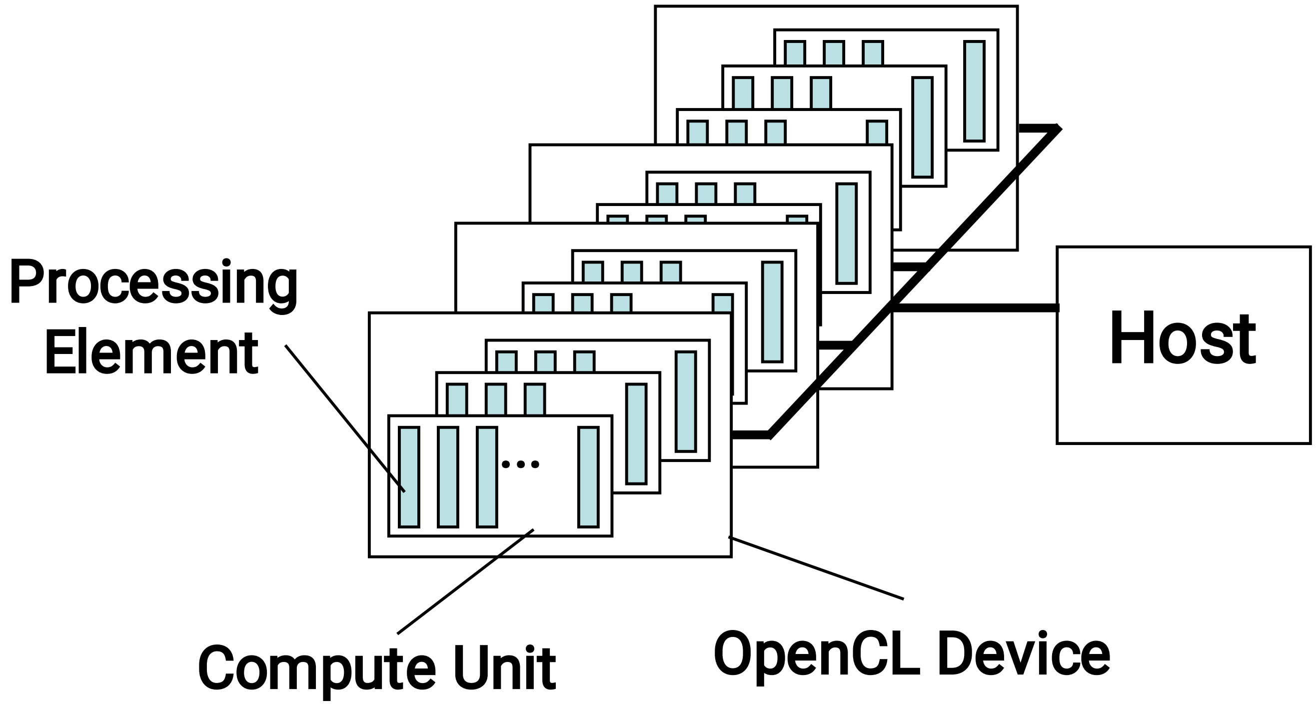
```
const cl::NDRange
      global(300, 200);

const cl::NDRange
      local(30, 20);

kernel(
  EnqueueArgs(global, local),
  …);
```

Indexing work

CUDA	OpenCL
gridDim	get_num_groups()
blockIdx	get_group_id()
blockDim	get_local_size()
gridDim * blockDim	get_global_size()
threadIdx	get_local_id()
blockIdx * blockdim + threadIdx	get_global_id()

IMPORTANT OPENCL CONCEPTS

OpenCL Platform Model



- One *Host* and one or more *OpenCL Devices*
  - Each OpenCL Device is composed of one or more *Compute Units*
    - Each Compute Unit is divided into one or more *Processing Elements*
- Memory divided into *host memory* and *device memory*

The BIG idea behind OpenCL

- Replace loops with functions (a kernel) executing at each point in a problem domain
  - E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

Traditional loops

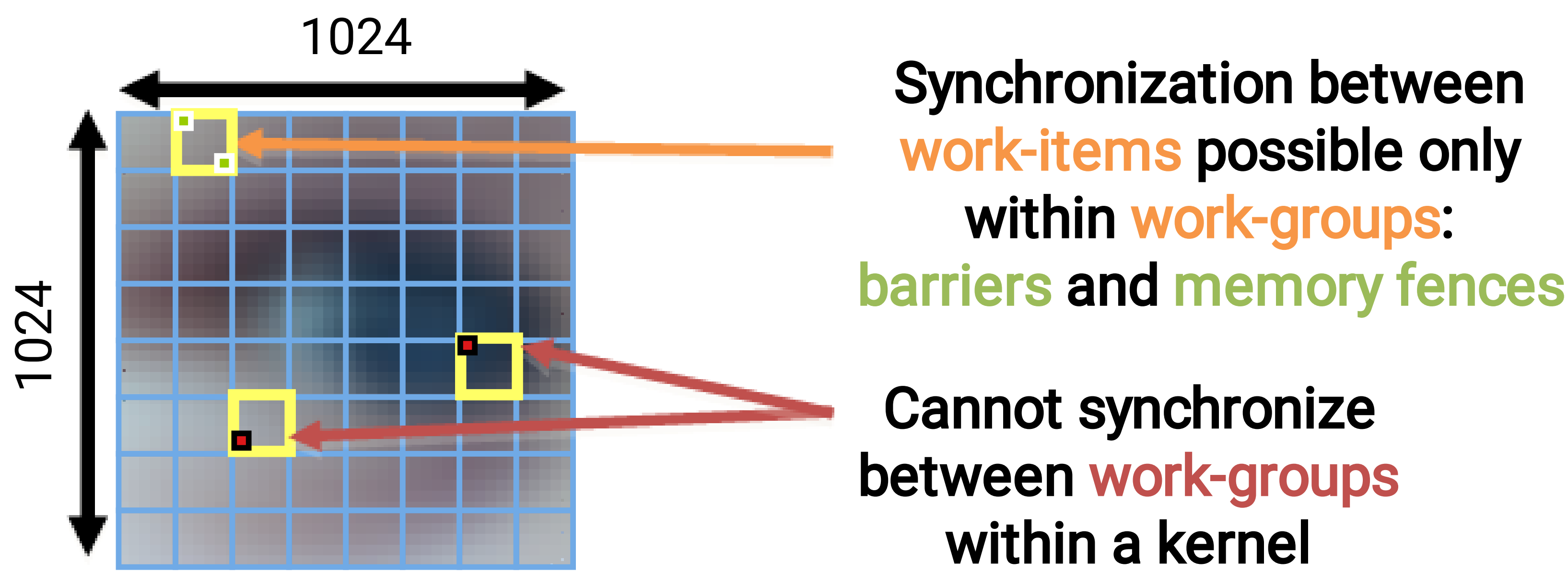
```
void
mul(const int n,
   const float *a,
   const float *b,
         float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
__kernel void
mul(__global const float *a,
   __global const float *b,
   __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
// many instances of the kernel,
// called work-items, execute
// in parallel
```

An N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (**work-group**, executes together)



- Choose the dimensions that are “best” for your algorithm

OpenCL N Dimensional Range (NDRange)

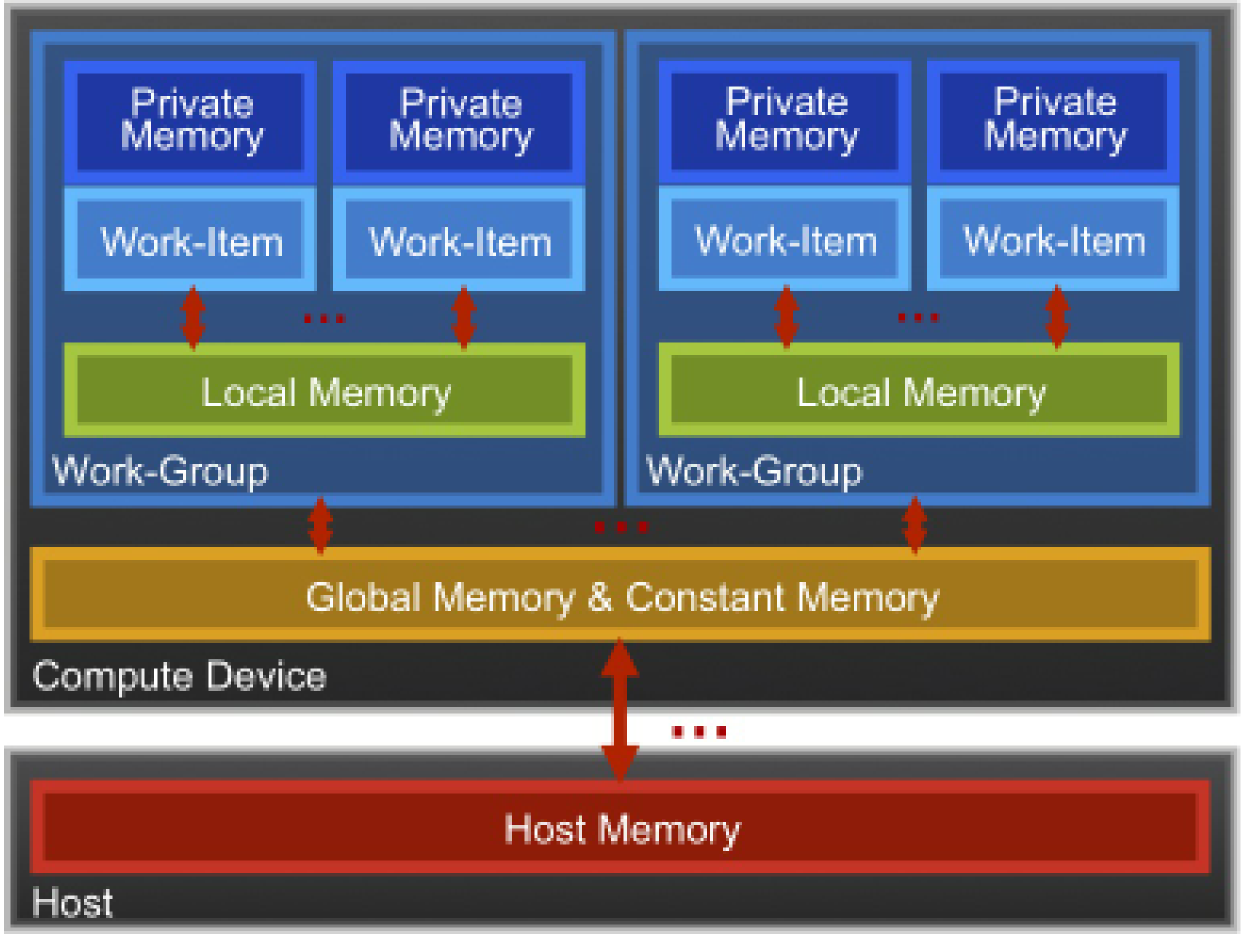
- The problem we want to compute should have some **dimensionality**;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension – this is called the **global** size
- We associate each point in the iteration space with a **work-item**

OpenCL N Dimensional Range (NDRange)

- Work-items are grouped into **work-groups**; work-items within a work-group can share **local memory** and can **synchronize**
- We can specify the number of work-items in a work-group – this is called the **local** (work-group) size
- Or the OpenCL run-time can choose the work-group size for you (usually not optimally)

OpenCL Memory model

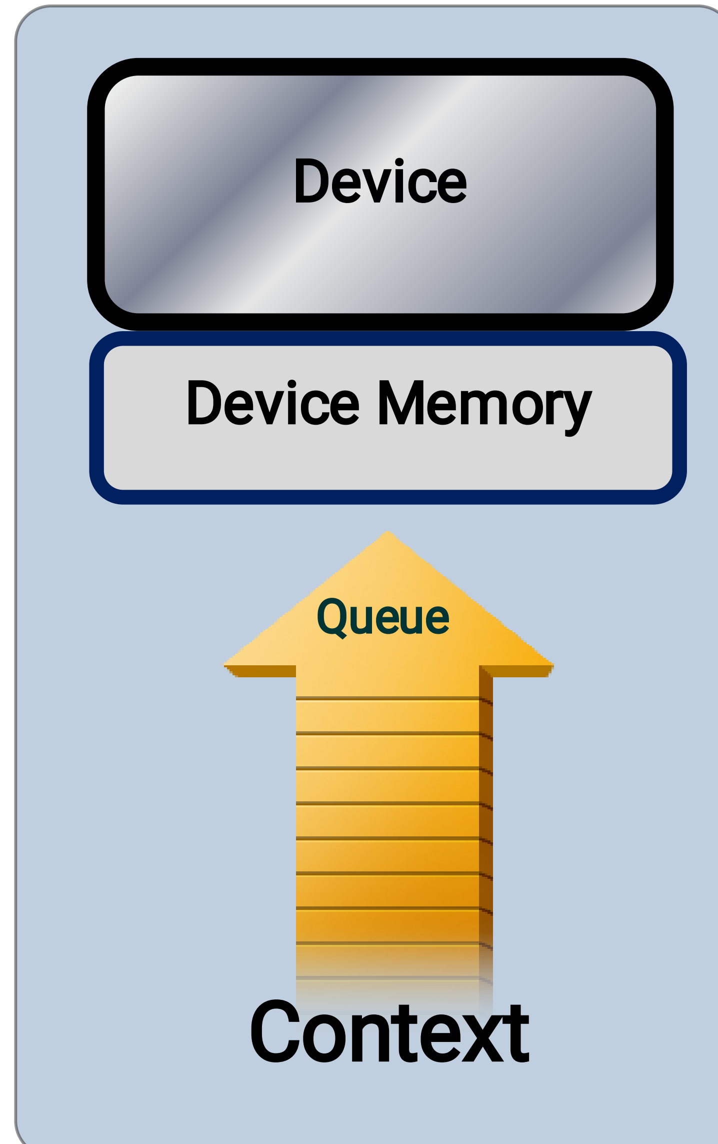
- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory / Constant Memory*
  - Visible to all work-groups
- *Host memory*
  - On the CPU



Memory management is explicit:
You are responsible for moving data from host → global → local *and* back

Context and Command-Queues

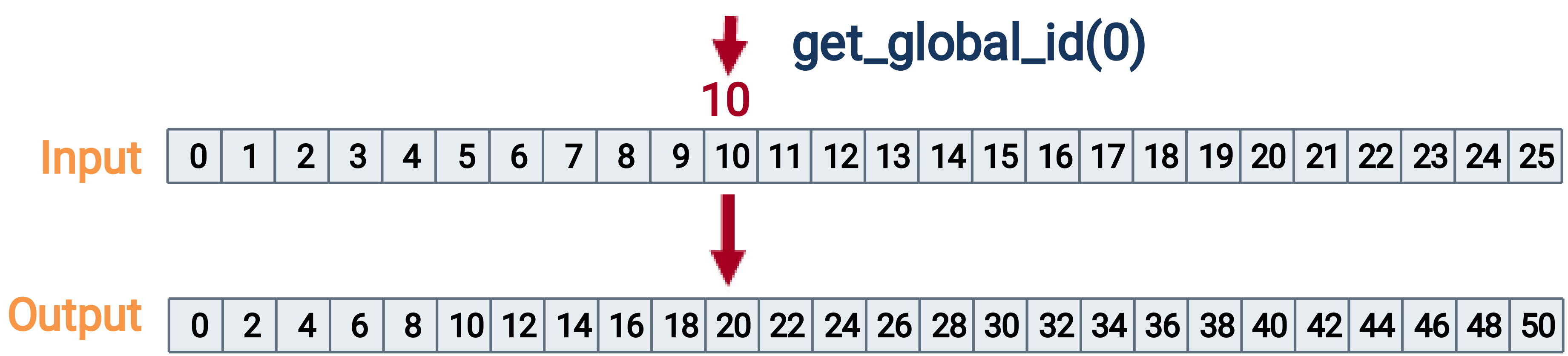
- *Context*:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The *context* includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All *commands* for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a *command-queue*.
- Each *command-queue* points to a single device within a context.



Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a **kernel** for each point in the domain

```
__kernel void times_two(
    __global float* input,
    __global float* output)
{
  int i = get_global_id(0);
  output[i] = 2.0f * input[i];
}
```



Example: vector addition

- The “hello world” program of data parallel programming is a program to add two vectors

$$C[i] = A[i] + B[i] \text{ for } i=0 \text{ to } N-1$$

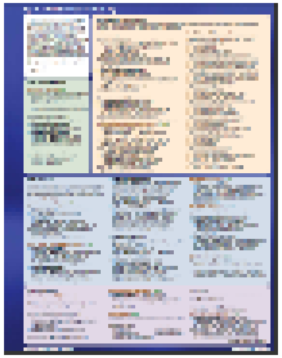
- For the OpenCL solution, there are two parts
  - Kernel code
  - Host code

Vector Addition - Kernel

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

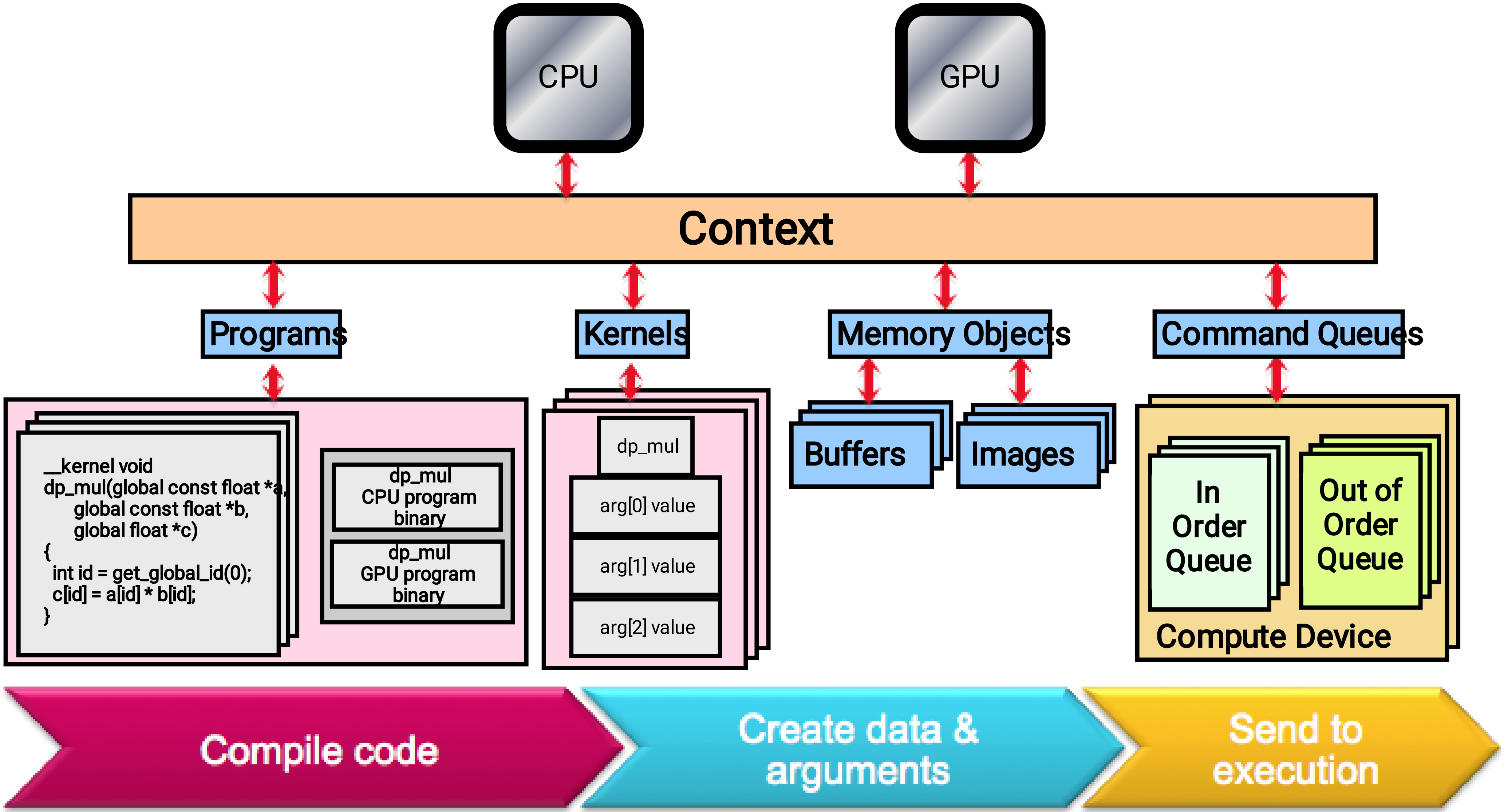
Vector Addition – Host

- The <u>host program</u> is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels
- 5 simple steps in a basic host program:
  1. Define the *platform* … platform = devices+context+queues
  2. Create and Build the *program* (dynamic library for kernels)
  3. Setup *memory* objects
  4. Define the *kernel* (attach arguments to kernel functions)
  5. Submit *commands* … transfer memory objects and execute kernels



As we go over the next set of slides, cross reference content on the slides to the reference card. This will help you get used to the reference card and how to pull information from the card and express it in code.

The basic platform and runtime APIs in OpenCL (using C)



1. Define the platform

- Grab the first available platform:

```
err = clGetPlatformIDs(1, &firstPlatformId,
                       &numPlatforms);
```

- Use the first CPU device the platform provides:

```
err = clGetDeviceIDs(firstPlatformId,
        CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
```

- Create a simple context with a single device:

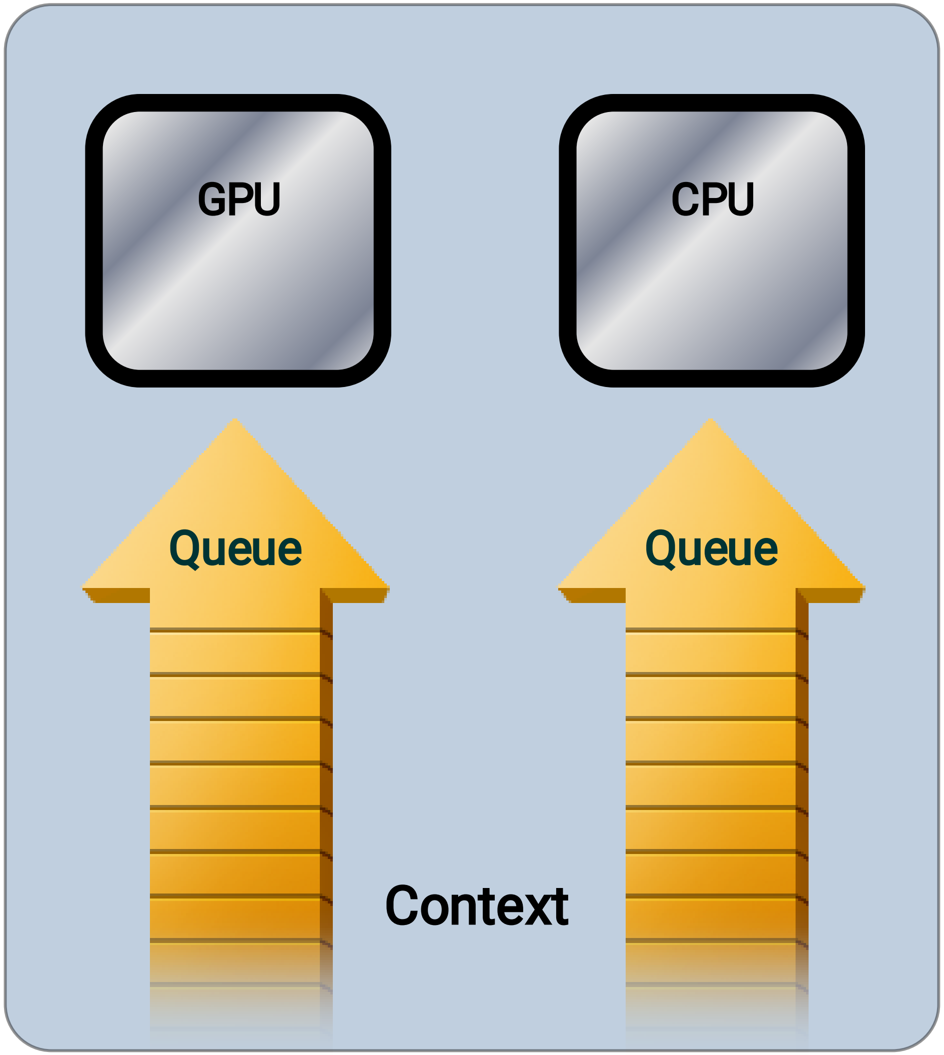
```
context = clCreateContext(firstPlatformId, 1,
                 &device_id, NULL, NULL, &err);
```

- Create a simple command-queue to feed our device:

```
commands = clCreateCommandQueue(context, device_id,
                                0, &err);
```

Command-Queues

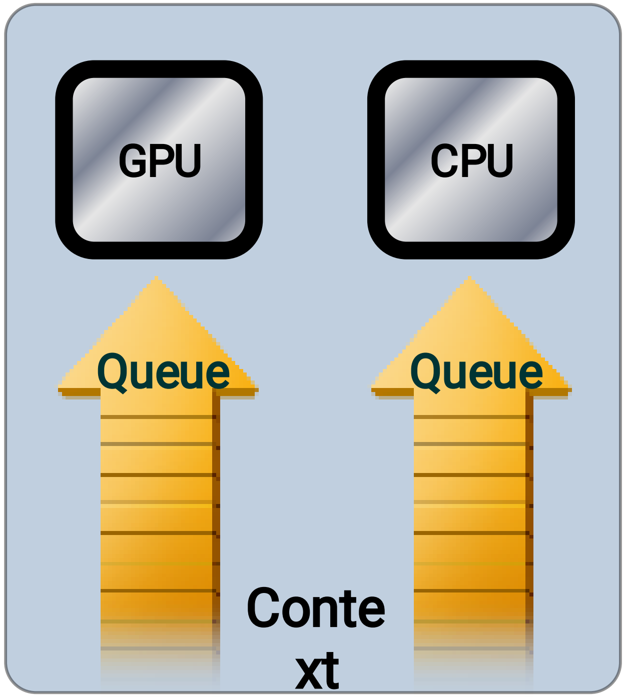
- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define independent streams of commands that don't require synchronization



Command-Queue execution details

Command queues can be configured in different ways to control how commands execute

- *In-order queues*:
  - Commands are enqueued and complete in the order they appear in the program (program-order)
- *Out-of-order queues*:
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.
- Execution of commands in the command-queue are guaranteed to be completed at synchronization points
  - Discussed later



2. Create and Build the program

- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (for real applications).

- Build the program object:

```
program = clCreateProgramWithSource(context, 1
        (const char**) &KernelSource, NULL, &err);
```

- Compile the program to create a “dynamic library” from which specific kernels can be pulled:

```
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);
```

Error messages

- Fetch and print error messages:

```
if (err != CL_SUCCESS) {
 size_t len;
 char buffer[2048];
 clGetProgramBuildInfo(program, device_id,
  CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
 printf(“%s\n”, buffer);
}
```

- Important to do check all your OpenCL API error messages!

- Easier in C++ with try/catch (see later)

3. Setup Memory Objects

- For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C.
- Create input vectors and assign values on the host:

```
float h_a[LENGTH], h_b[LENGTH], h_c[LENGTH];
for (i = 0; i < length; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
}
```

- Define OpenCL memory objects:

```
d_a = clCreateBuffer(context, CL_MEM_READ_ONLY,
                sizeof(float)*count, NULL, NULL);
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY,
                sizeof(float)*count, NULL, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                sizeof(float)*count, NULL, NULL);
```

What do we put in device memory?

Memory Objects:

- A handle to a reference-counted region of global memory.

There are two kinds of memory object

- *Buffer* object:
  - Defines a linear collection of bytes ("*just a C array*").
  - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
- *Image* object:
  - Defines a two- or three-dimensional region of memory.
  - Image data can only be accessed with read and write functions, i.e. these are opaque data structures. The read functions use a sampler.

Used when interfacing with a graphics API such as OpenGL. We won't use image objects in this tutorial.

Creating and manipulating buffers

- Buffers are declared on the host as type: cl_mem

- Arrays in host memory hold your original host-side data:

```
float h_a[LENGTH], h_b[LENGTH];
```

- Create the buffer (d_a), assign sizeof(float)*count bytes from “h_a” to the buffer and copy it into device memory:

```
cl_mem d_a = clCreateBuffer(context,
  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   sizeof(float)*count, h_a, NULL);
```

Conventions for naming buffers

- It can get confusing about whether a host variable is just a regular C array or an OpenCL buffer
- A useful convention is to prefix the names of your regular **h**ost C arrays with "**h_**" and your OpenCL buffers which will live on the **d**evice with "**d_**"

Creating and manipulating buffers

- Other common memory flags include:
  CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE
- These are from the point of view of the **device**
- Submit command to copy the buffer back to host memory at “h_c”:
  - CL_TRUE = blocking, CL_FALSE = non-blocking

```
clEnqueueReadBuffer(queue, d_c, CL_TRUE,
      sizeof(float)*count, h_c,
    NULL, NULL, NULL);
```

4. Define the kernel

- Create kernel object from the kernel function “vadd”:

```
kernel = clCreateKernel(program, “vadd”, &err);
```

- Attach arguments of the kernel function “vadd” to memory objects:

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int),          &count);
```

5. Enqueue commands

- Write Buffers from host into global memory (as non-blocking operations):

```
err = clEnqueueWriteBuffer(commands, d_a, CL_FALSE,
       0, sizeof(float)*count, h_a, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, d_b, CL_FALSE,
       0, sizeof(float)*count, h_b, 0, NULL, NULL
```

- Enqueue the kernel for execution (note: in-order so OK):

```
err = clEnqueueNDRangeKernel(commands, kernel, 1,
           NULL, &global, &local, 0, NULL, NULL);
```

5. Enqueue commands

- Read back result (as a blocking operation). We have an in-order queue which assures the previous commands are completed before the read can begin.

```
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE,
        sizeof(float)*count, h_c, 0, NULL, NULL);
```

Vector Addition – Host Program

```
// create the OpenCL context on a GPU device
cl_context 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);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

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

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                  sizeof(cl_float)*n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
                   &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
               sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
               sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
               sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
            global_work_size, NULL,0,NULL,NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
               CL_TRUE, 0,
               n*sizeof(cl_float), dst,
               0, NULL, NULL);
```

Vector Addition – Host Program

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
            CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetC
```

Define platform and queues

```
cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

// allocate the buffer memory objects
memobjs
    CL_M
```

Define memory objects

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

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                sizeof(cl_float)*n, NULL, NULL);

// create the p
program = clC
                &program_source, NULL, NULL);
```

Create the program

```
// build the progra
err = clBuildProgra
```

Build the program

```
// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],

err |= clSetK
                sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;
```

Create and setup kernel

```
// execute kernel
err = clEnqueue
            global
```

Execute the kernel

```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],

            0, NULL, NULL);
```

Read results on the host

It's complicated, but most of this is "boilerplate" and not as bad as it looks.

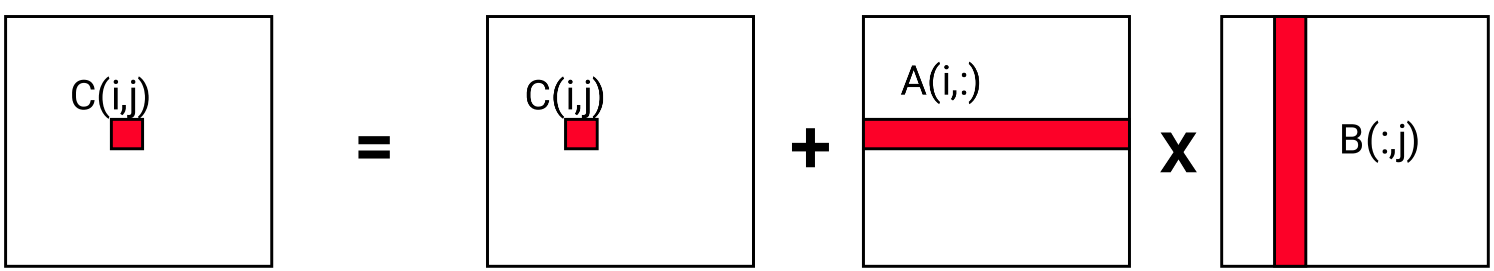
Exercise 2: Running the Vadd kernel

- Goal:
  - To inspect and verify that you can run an OpenCL kernel
- Procedure:
  - Take the provided C Vadd program. It will run a simple kernel to add two vectors together.
  - Look at the host code and identify the API calls in the host code. Compare them against the API descriptions on the OpenCL reference card.
  - There are some helper files which time the execution, output device information neatly and check errors.
- Expected output:
  - A message verifying that the vector addition completed successfully

Matrix multiplication: sequential code

We calculate C=AB, dimA = (N x P), dimB=(P x M), dimC=(N x M)

```
void mat_mul(int Mdim, int Ndim, int Pdim,
             float *A, float *B, float *C)
{
   int i, j, k;
   for (i = 0; i < Ndim; i++) {
      for (j = 0; j < Mdim; j++) {
         for (k = 0; k < Pdim; k++) {
            // C(i, j) = sum(over k) A(i,k) * B(k,j)
            C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
         }
      }
   }
}
```



Dot product of a row of A and a column of B for each element of C

Matrix multiplication performance

- Serial C code on CPU (single core).

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A

Device is Intel® Xeon® CPU, E5649 @ 2.53GHz
using the gcc compiler.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication: sequential code

```
void mat_mul(int Mdim, int Ndim, int Pdim,
                  float *A, float *B, float *C)
{
   int i, j, k;
   for (i = 0; i < Ndim; i++) {
      for (j = 0; j < Mdim; j++) {
         for (k = 0; k < Pdim; k++) {
            // C(i, j) = sum(over k) A(i,k) * B(k,j)
            C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
         }
      }
   }
}
```

We turn this into an OpenCL kernel!

Matrix multiplication: OpenCL kernel (1/2)

```
__kernel void mat_mul(
 const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
{
   int i, j, k;
   for (i = 0; i < Ndim; i++) {
      for (j = 0; j < Mdim; j++) {
         // C(i, j) = sum(over k) A(i,k) * B(k,j)
         for (k = 0; k < Pdim; k++) {
            C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
         }
      }
   }
}
```

Mark as a kernel function and specify memory qualifiers

Matrix multiplication: OpenCL kernel (2/2)

```
__kernel void mat_mul(
  const int Mdim, const int Ndim, const int Pdim,
  __global float *A, __global float *B, __global float *C)
{
   int i, j, k;
   i = get_global_id(0);
   j = get_global_id(1);
       for (k = 0; k < Pdim; k++) {
          // C(i, j) = sum(over k) A(i,k) * B(k,j)
          C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
       }

   }
}
```

Remove outer loops and set work-item co-ordinates

Matrix multiplication: OpenCL kernel

```
__kernel void mat_mul(
 const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
{
   int i, j, k;
   i = get_global_id(0);
   j = get_global_id(1);
   // C(i, j) = sum(over k) A(i,k) * B(k,j)
   for (k = 0; k < Pdim; k++) {
     C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
   }
}
```

Matrix multiplication: OpenCL kernel improved

Rearrange and use a local scalar for intermediate C element values
(a common optimization in Matrix Multiplication functions)

```
__kernel void mmul(
   const int Mdim,
   const int Ndim,
   const int Pdim,
   __global float *A,
   __global float *B,
   __global float *C)
{
  int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tmp = 0.0f;
  for (k = 0; k < Pdim; k++)
   tmp += A[i*Ndim+k]*B[k*Pdim+j];
  }
  C[i*Ndim+j] += tmp;
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9

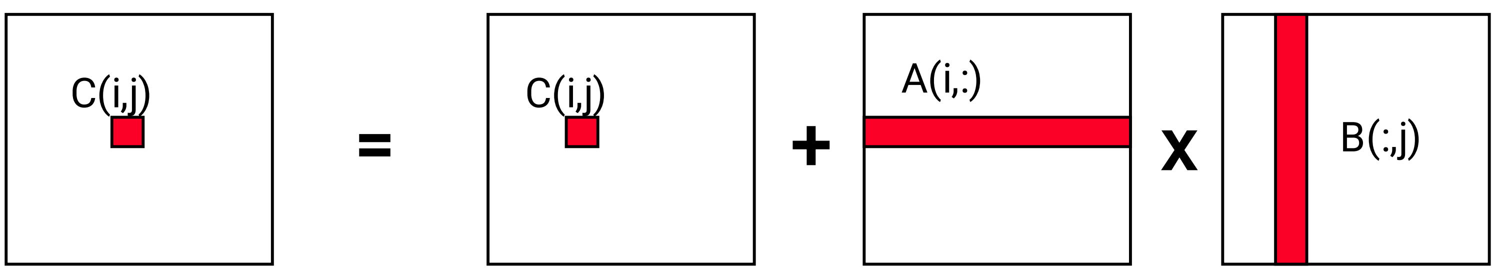
Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

UNDERSTANDING THE OPENCL MEMORY HIERARCHY

Optimizing matrix multiplication

- MM cost determined by FLOPS and memory movement:
  - $2*n^3 = O(n^3)$ FLOPS
  - Operates on $3*n^2 = O(n^2)$ numbers
- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOPS as possible.
- Outer product algorithms are faster, but for pedagogical reasons, let's stick to the simple dot-product algorithm.

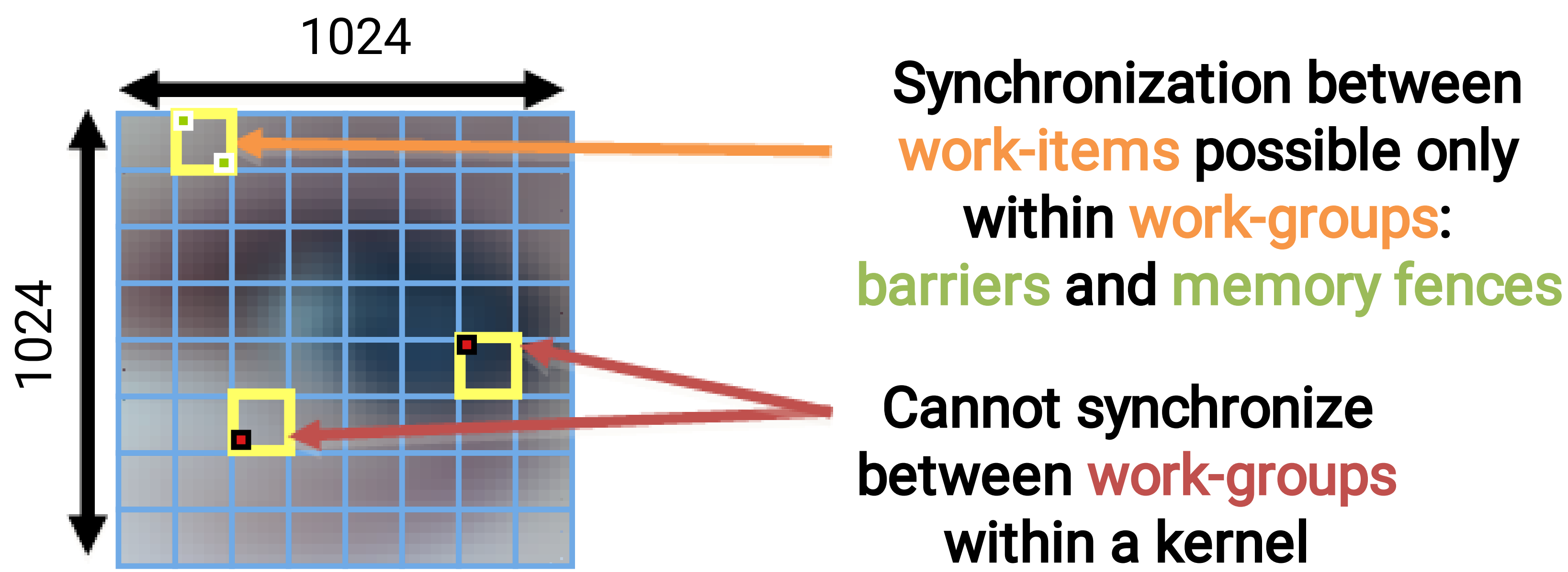


Dot product of a row of A and a column of B for each element of C

- We will work with work-item/work-group sizes and the memory model to optimize matrix multiplication

An N-dimensional domain of work-items

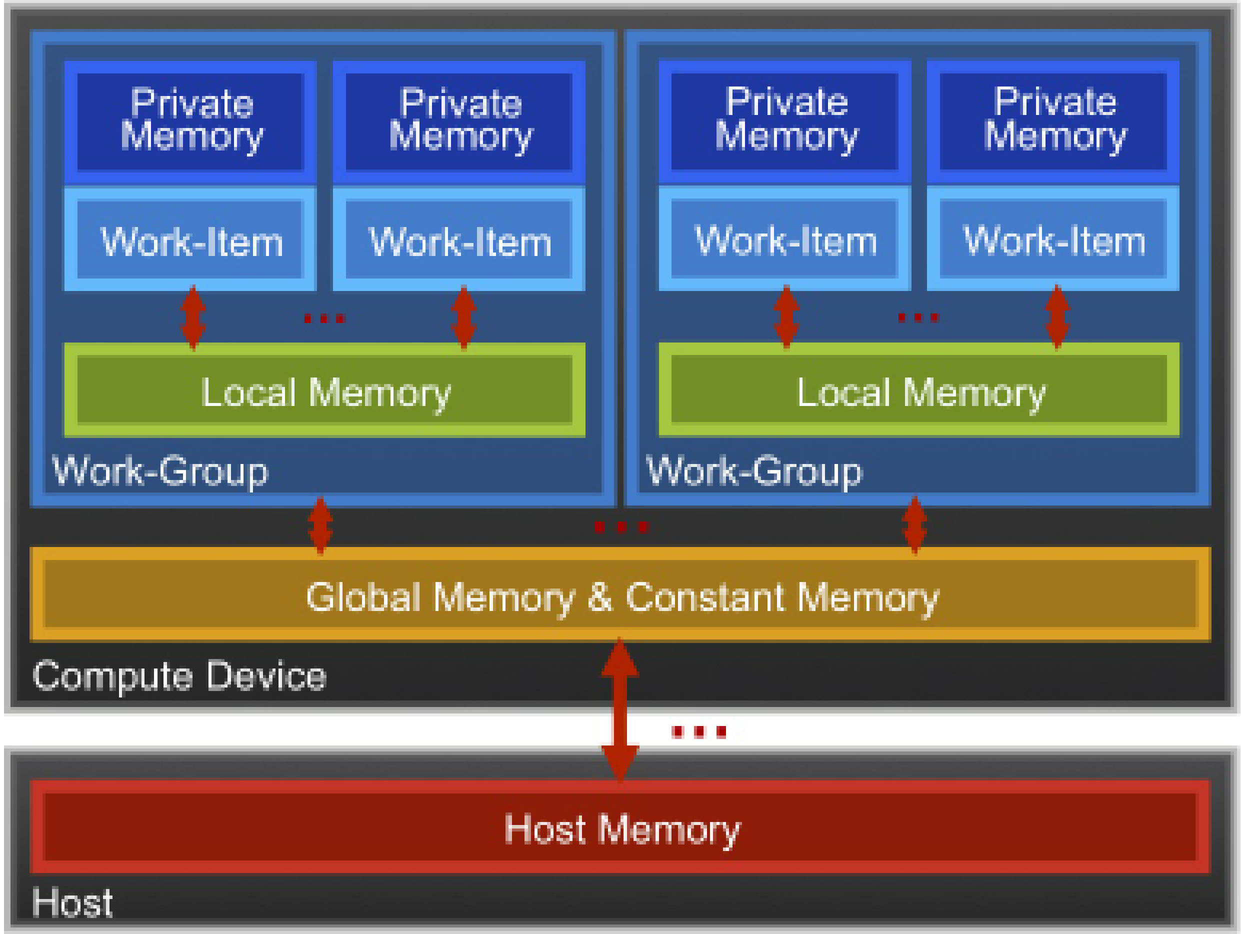
- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (work-group, executes together)



- Choose the dimensions that are “best” for your algorithm

OpenCL Memory model

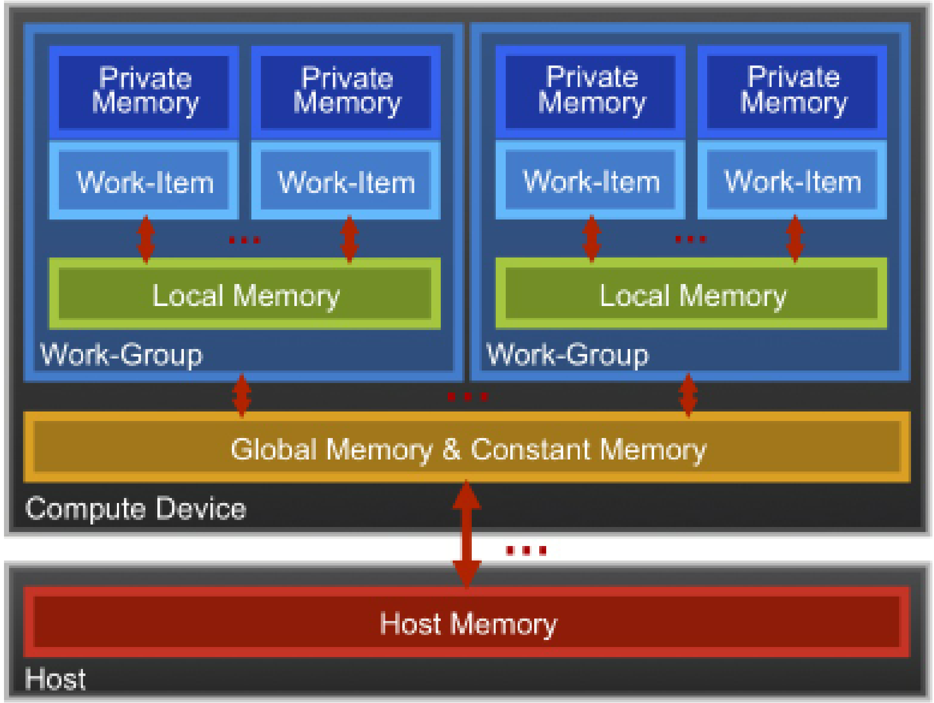
- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU



Memory management is **explicit**:
You are responsible for moving data from host → global → local *and* back

OpenCL Memory model

- Private Memory
  - Fastest & smallest: O(10) words/ WI
- Local Memory
  - Shared by all WI's in a work-group
  - But not shared between work-groups!
  - O(1-10) Kbytes per work-group
- Global/Constant Memory
  - O(1-10) Gbytes of Global memory
  - O(10-100) Kbytes of Constant memory
- Host memory
  - On the CPU - GBytes



Memory management is **explicit**:
O(1-10) Gbytes/s bandwidth to discrete GPUs for Host <-> Global transfers

Private Memory

- Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance

- Private Memory:
  - A **very scarce** resource, only a few tens of 32-bit words per Work-Item at most
  - If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance*
  - Think of these like registers on the CPU

* Occupancy on a GPU

Local Memory*

- Tens of KBytes per Compute Unit
  - As multiple Work-Groups will be running on each CU, this means only a fraction of the total Local Memory size is available to each Work-Group
- Assume O(1-10) KBytes of Local Memory per Work-Group
  - Your kernels are responsible for transferring data between Local and Global/Constant memories … there are optimized library functions to help
  - E.g. async_work_group_copy(), async_workgroup_strided_copy(), …
- Use Local Memory to hold data that can be reused by all the work-items in a work-group
- Access patterns to Local Memory affect performance in a similar way to accessing Global Memory
  - Have to think about things like coalescence & bank conflicts

* Typical figures for a 2013 GPU

Local Memory

- Local Memory doesn't always help…
  - CPUs don't have special hardware for it
  - This can mean excessive use of Local Memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
  - So, your mileage may vary!