OPENCL

Episode 2 - OpenCL Fundamentals

David W. Gohara, Ph.D.
Center for Computational Biology
Washington University School of Medicine, St. Louis
email: sdg0919@gmail.com twitter: iGotchi

MacResearch
http://www.macresearch.org

THANK YOU

MACCIUS™
Application Hosting. Managed Services.

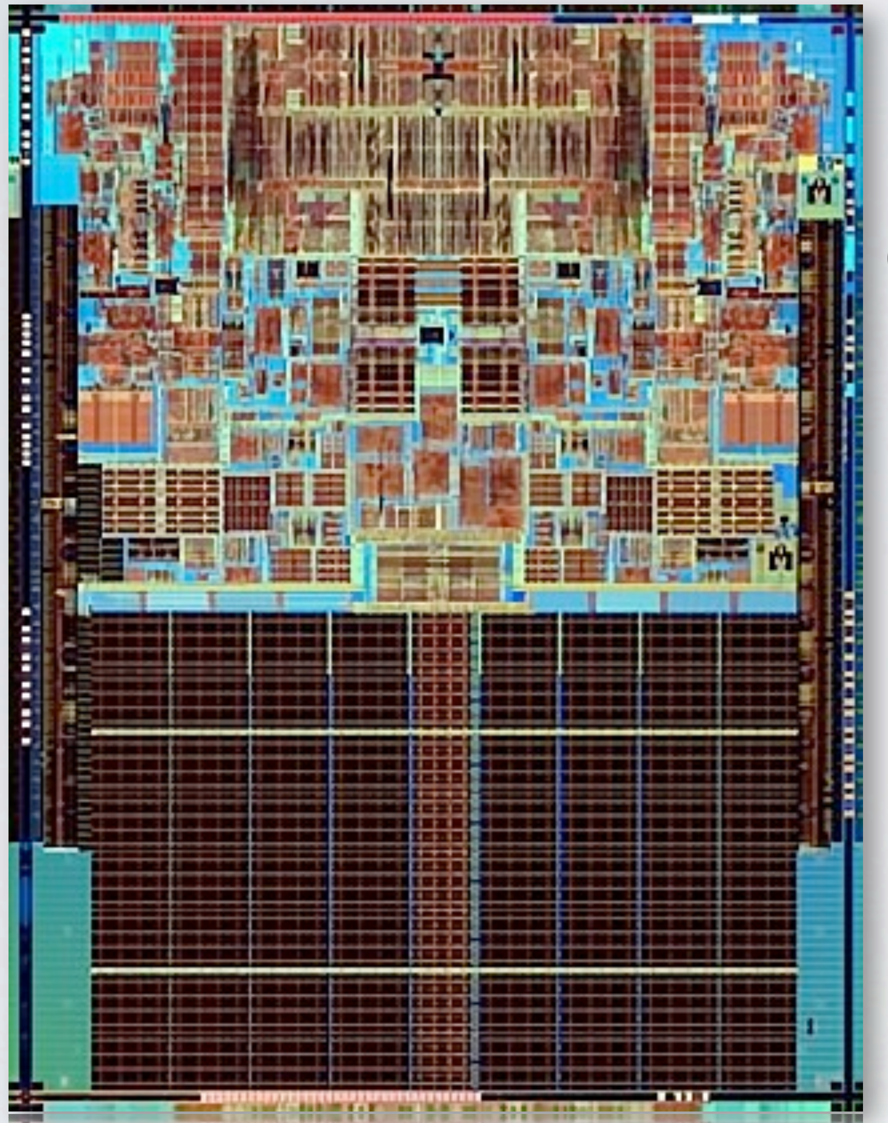
SUPPORTED GRAPHICS CARDS

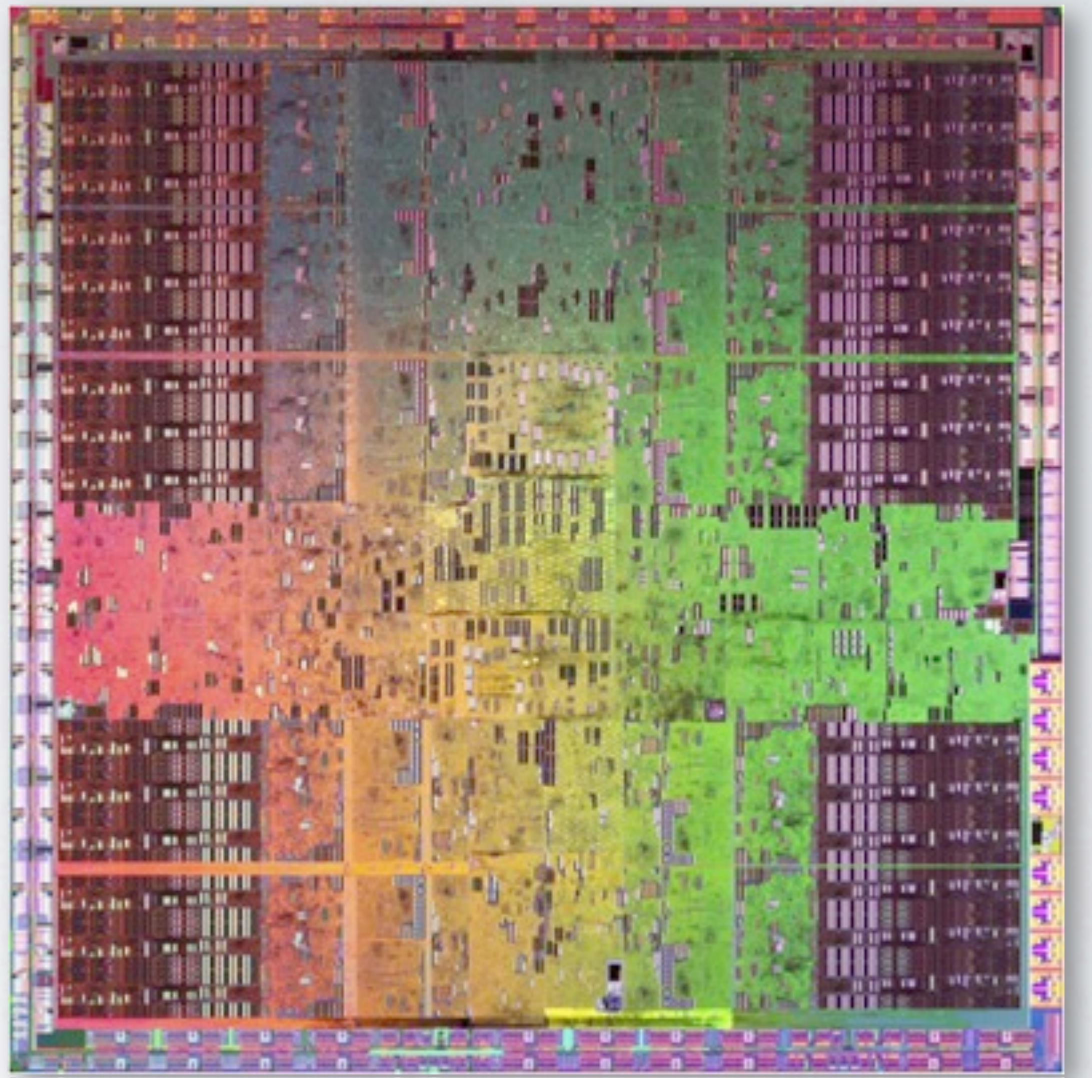
- NVIDIA GeForce 9400M
- GeForce 9600M GT
- GeForce 8600M GT
- GeForce GT 120
- GeForce GT 130
- GeForce GTX 285
- GeForce 8800 GT
- GeForce 8800 GS
- Quadro FX 4800
- Quadro FX5600

- ATI Radeon 4850
- Radeon 4870

http://www.apple.com/macosx/specs.html

Q & A


Core 2 Duo


NVIDIA GT200

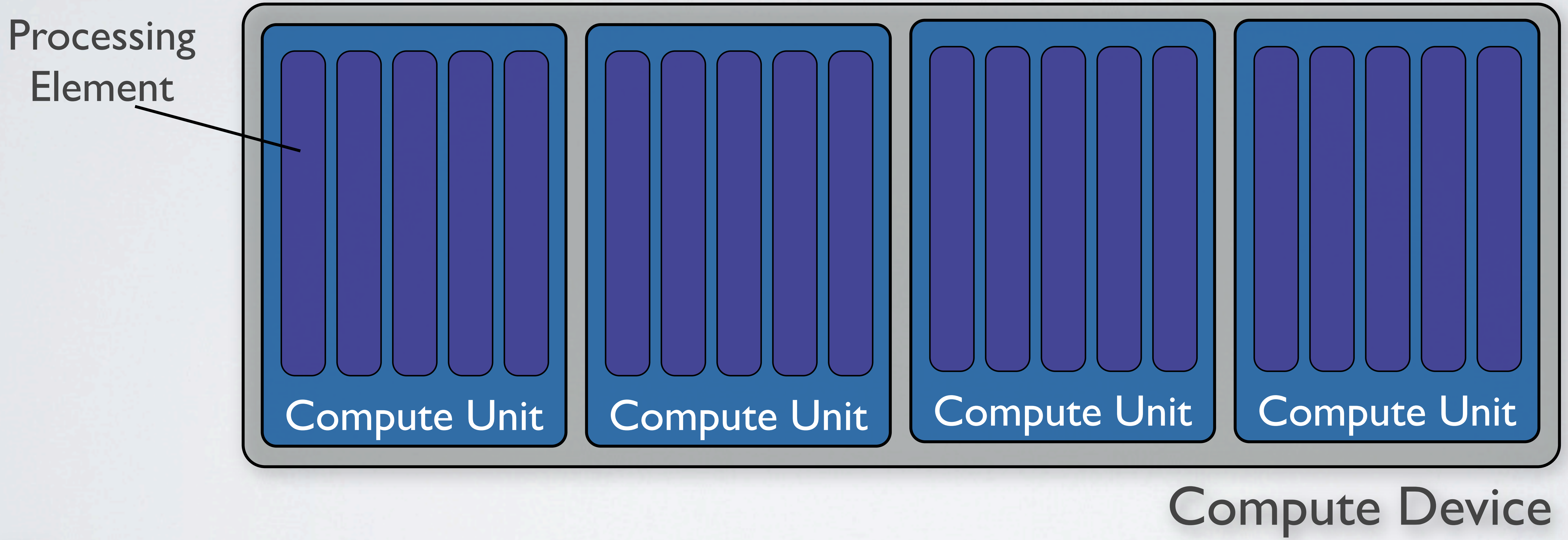
MacResearch
http://www.macresearch.org

OPENCL OBJECTS

- Compute devices
- Memory objects
  - Arrays
  - Images
- Executable objects
  - Compute program
  - Compute kernel

OPENCL OBJECTS - DEVICES

- A processor of some kind that executes data-parallel programs

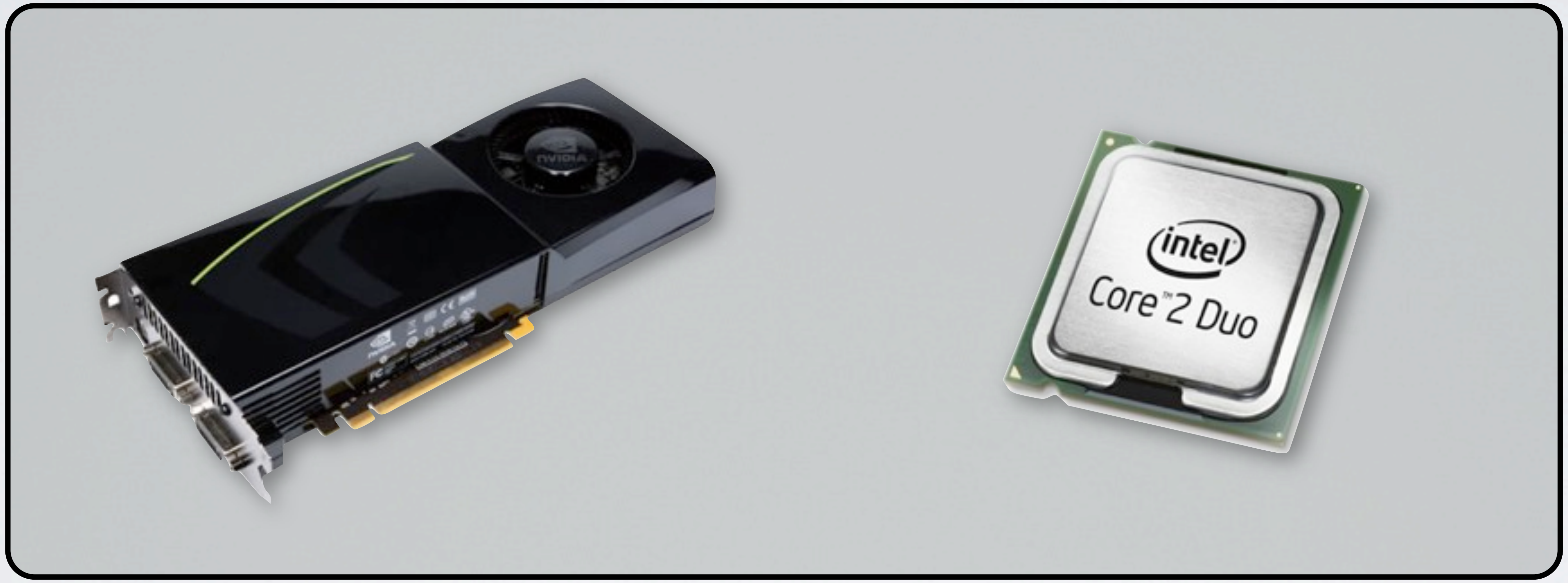


OPENCL OBJECTS - DEVICES

- A processor of some kind that executes data-parallel programs

OPENCL OBJECTS - DEVICES

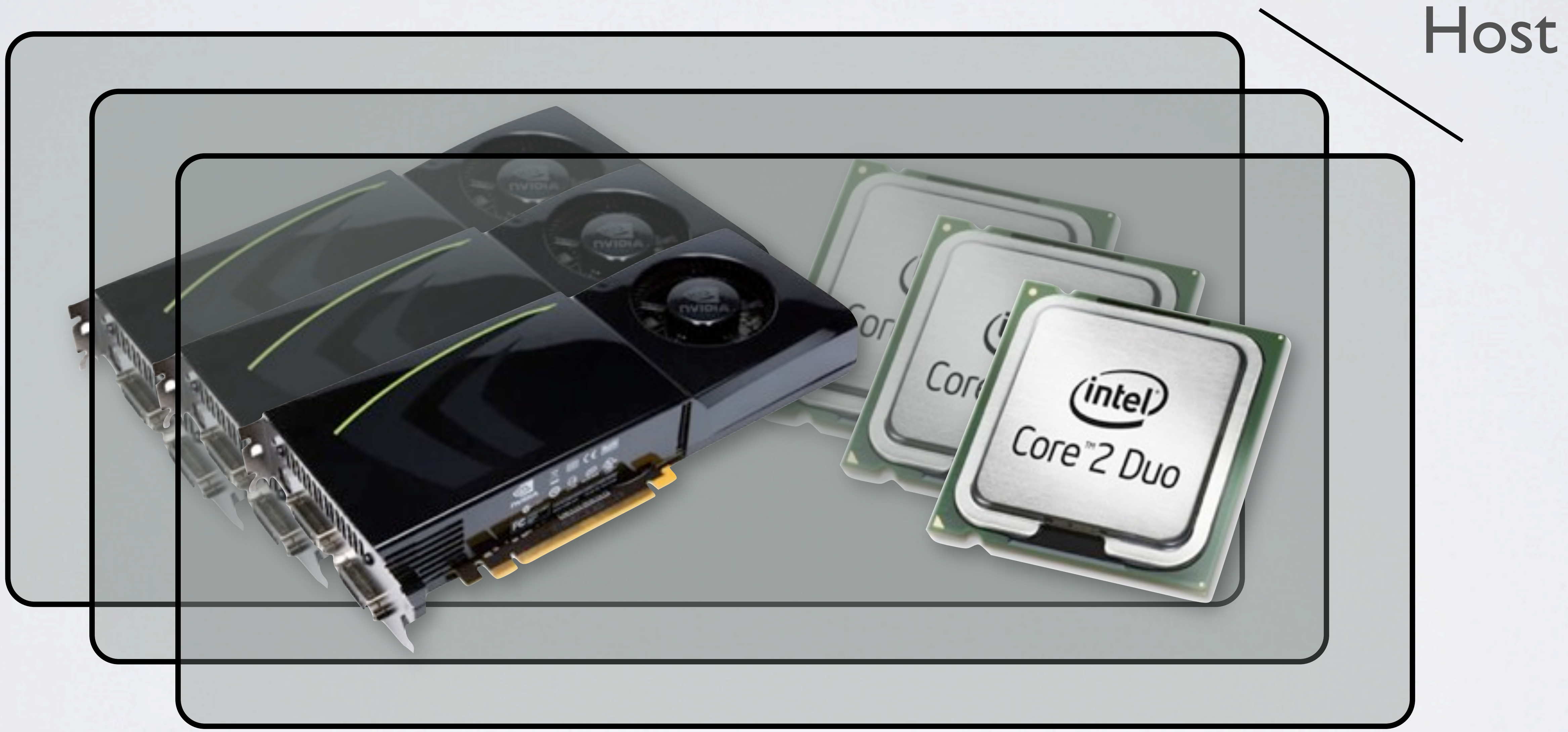
- A processor of some kind that executes data-parallel programs



Device Group

OPENCL OBJECTS - DEVICES

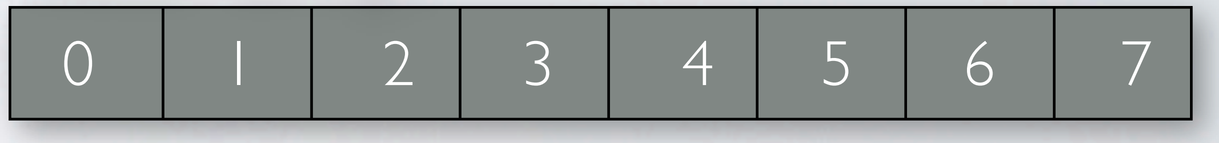
- A group of devices are contained in a **host**



OPENCL OBJECTS - MEMORY

- Arrays
  - Work exactly like arrays in C
  - Address elements via a pointer
  - Array reads/writes on the CPU are cached
  - Array reads/writes on the GPU are usually not

float *array;



float element = array[2];

element == 2

OPENCL OBJECTS - MEMORY

- Images
  - 2D and 3D images
  - Image data is stored in an optimized non-linear format
    - Elements are not directly accessed via pointers
  - Data reads use the texture cache

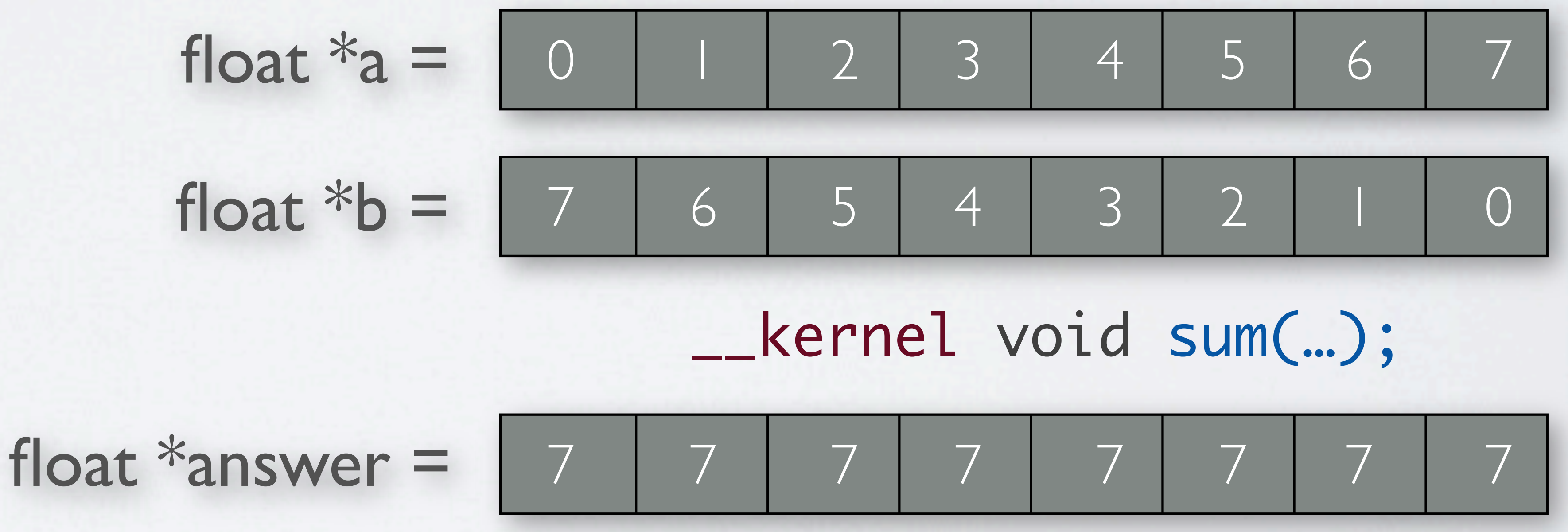
2D Image

3D Image

OPENCL OBJECTS - EXECUTABLES

- Compute kernel
  - A data-parallel function that is executed by the compute object (CPU or GPU)

```
__kernel void
sum(__global const float *a,
    __global const float *b,
    __global float *answer)
{
  int xid = get_global_id(0);
  answer[xid] = a[xid] + b[xid];
}
```


float *a = 0 1 2 3 4 5 6 7
float *b = 7 6 5 4 3 2 1 0
__kernel void sum(…);
float *answer = 7 7 7 7 7 7 7 7

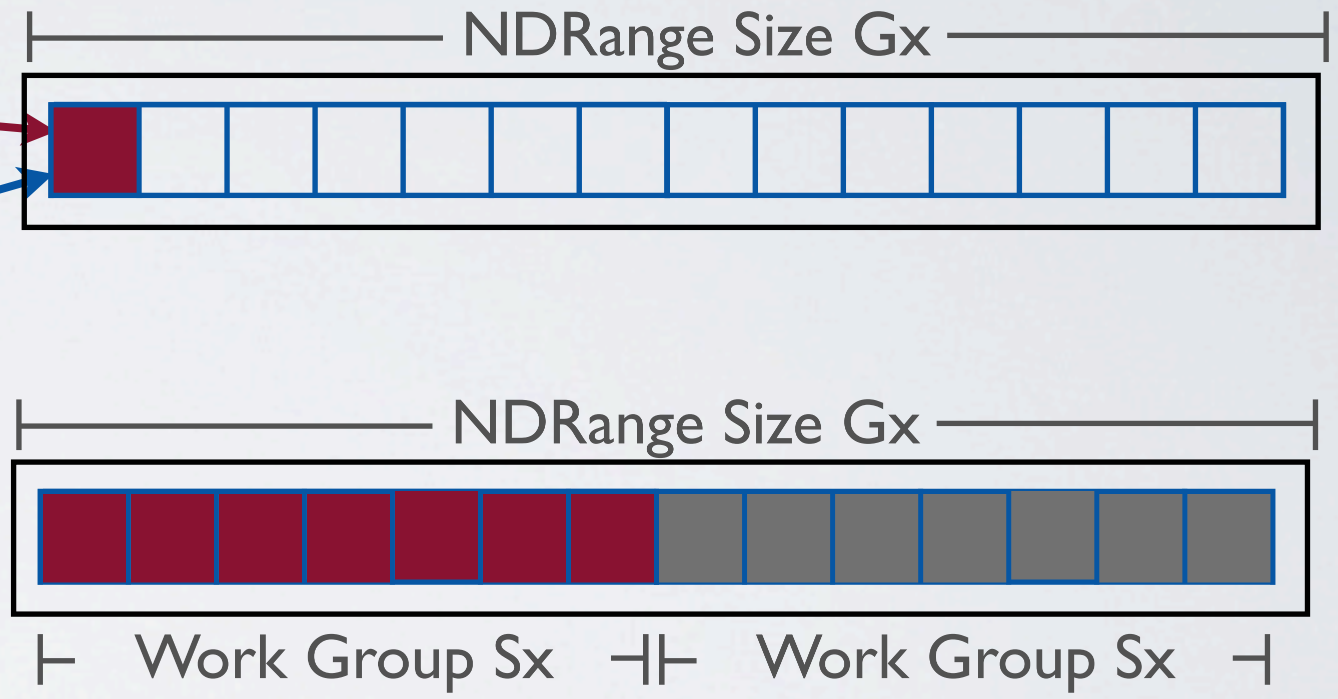
OPENCL OBJECTS - EXECUTABLES

- Compute program
    - A group of compute kernels and functions

```
__kernel void sub{...}

__kernel void transpose{...}

float cross_product{...}

...

__kernel void fft_radix2{...}
```

OPENCL WORK UNITS

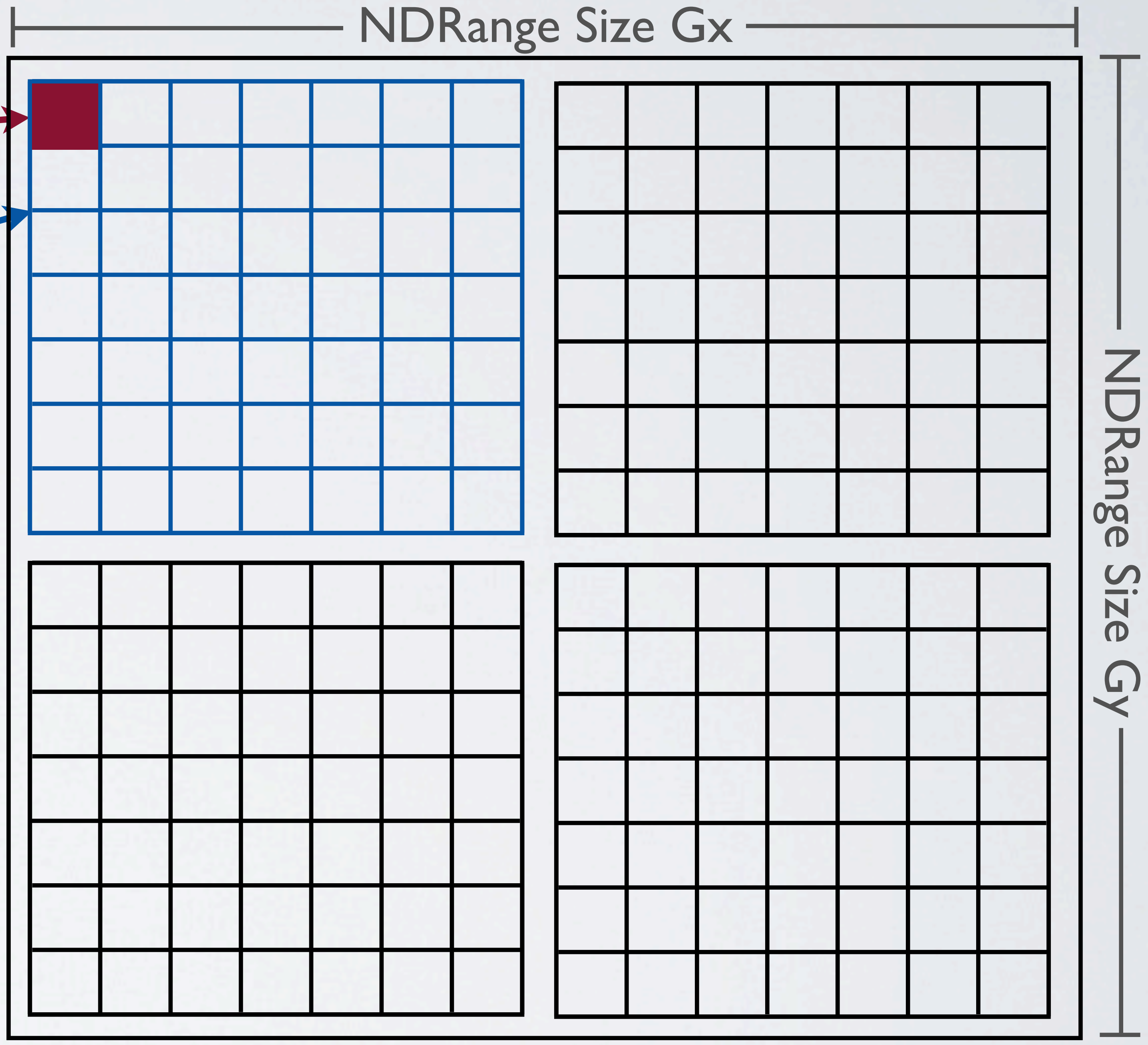
- A unit of work is called a **work-item**
- Work items are grouped into a **work-group**
- In CUDA a work-item is a CUDA thread
- In CUDA a work-group is a CUDA thread **block**



NDRange Size = Global Size
Work Group Size = Local Size

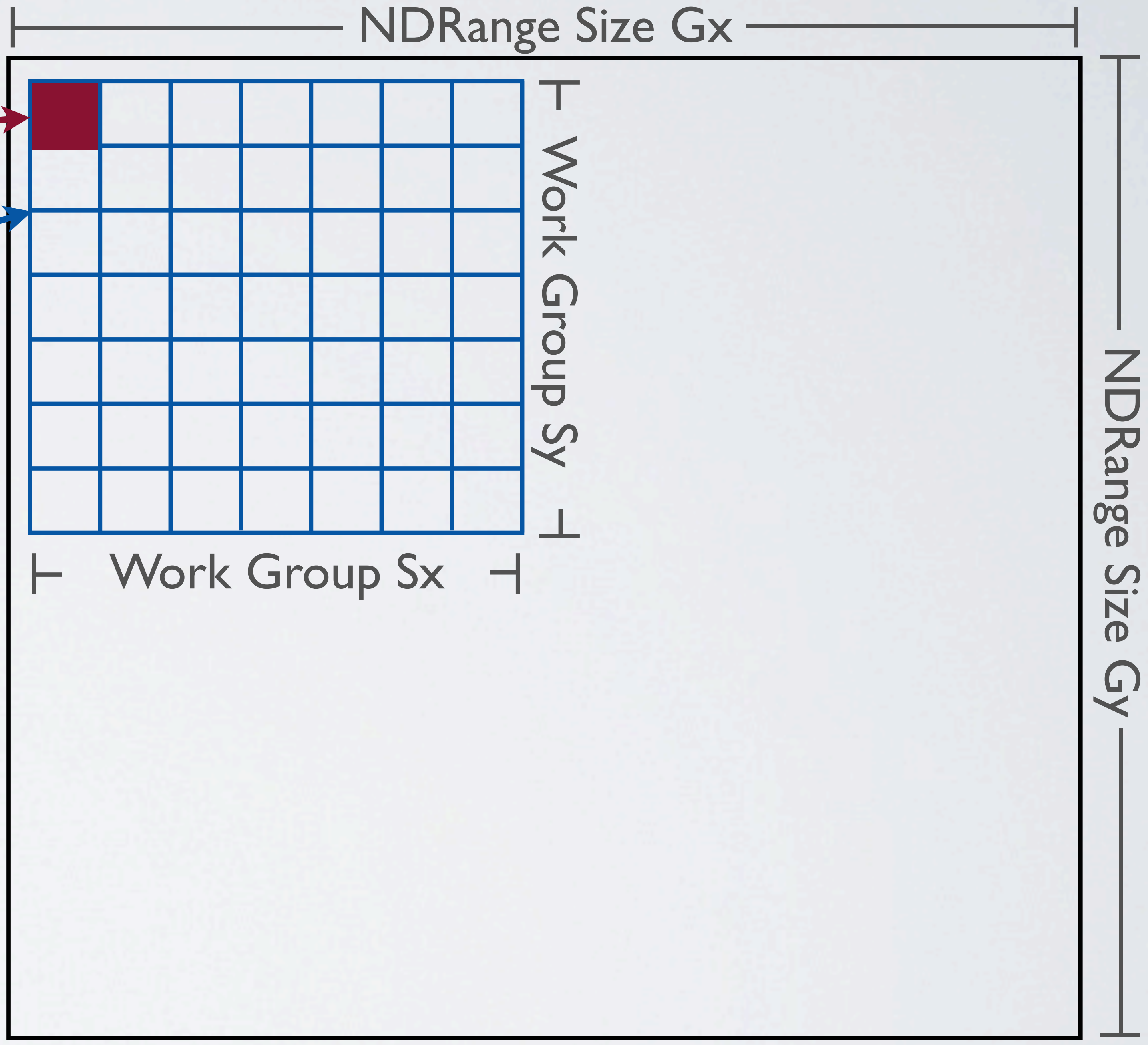
OPENCL WORK UNITS

- A unit of work is called a **work-item**
- Work items are grouped into a **work-group**
- In CUDA a work-item is a CUDA thread
- In CUDA a work-group is a CUDA thread **block**



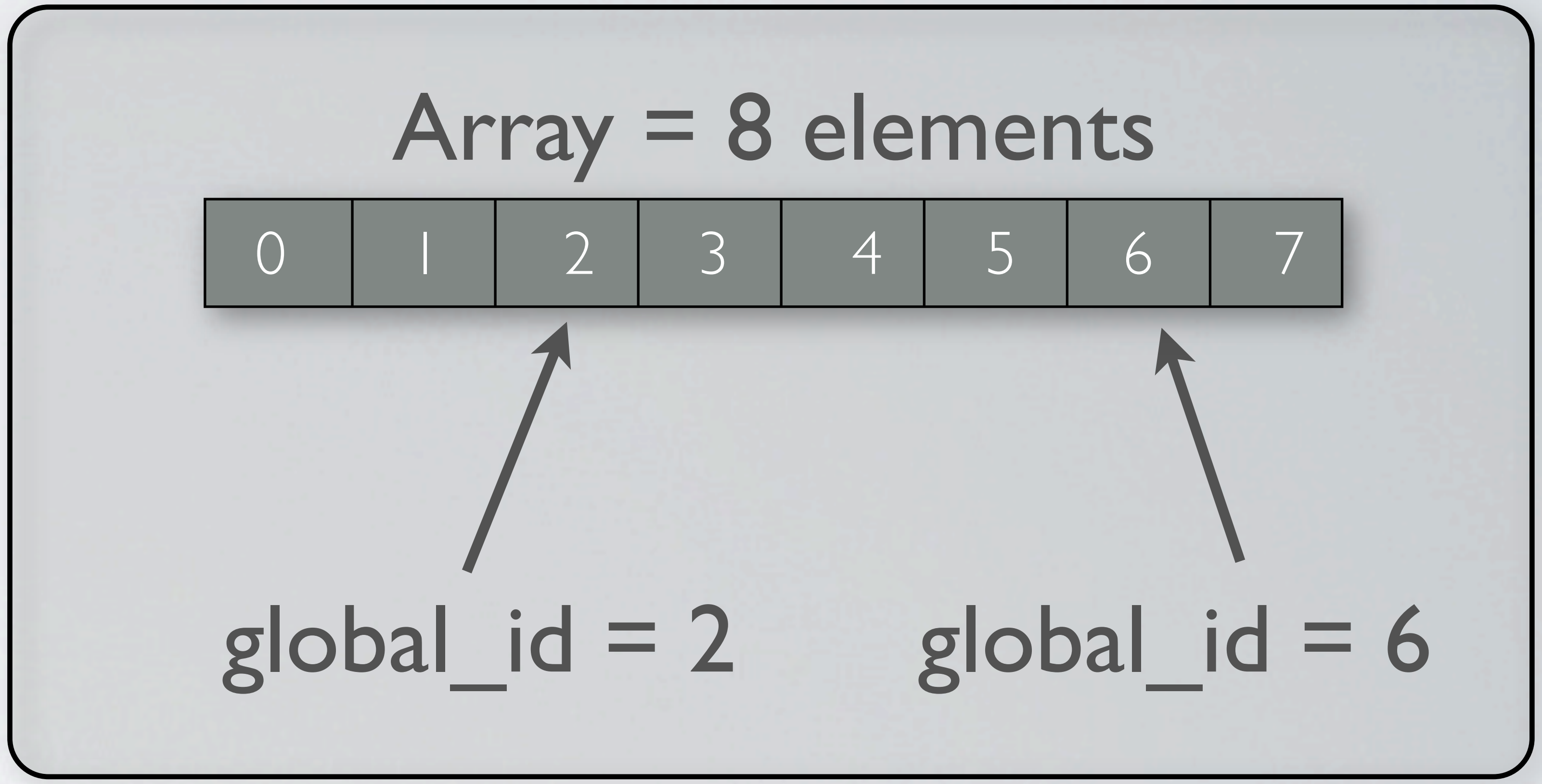
OPENCL WORK UNITS

- A unit of work is called a **work-item**
- Work items are grouped into a **work-group**
- In CUDA a work-item is a CUDA thread
- In CUDA a work-group is a CUDA thread **block**



WORK-ITEM IDENTIFIERS

- Each work-item is "aware" of what element of a problem it is working on
- Each work-item (and work-group) can be identified within the kernel
- The entire range of work-items is defined by the **NDRange**



```
size_t get_local_id(x);
size_t get_global_id(x);
```

where x = 0, 1 or 2

OPENCL KERNELS

- Basically the C programming language with some additions
  - 2D and 3D image types
  - Built-in methods
  - Vector data types

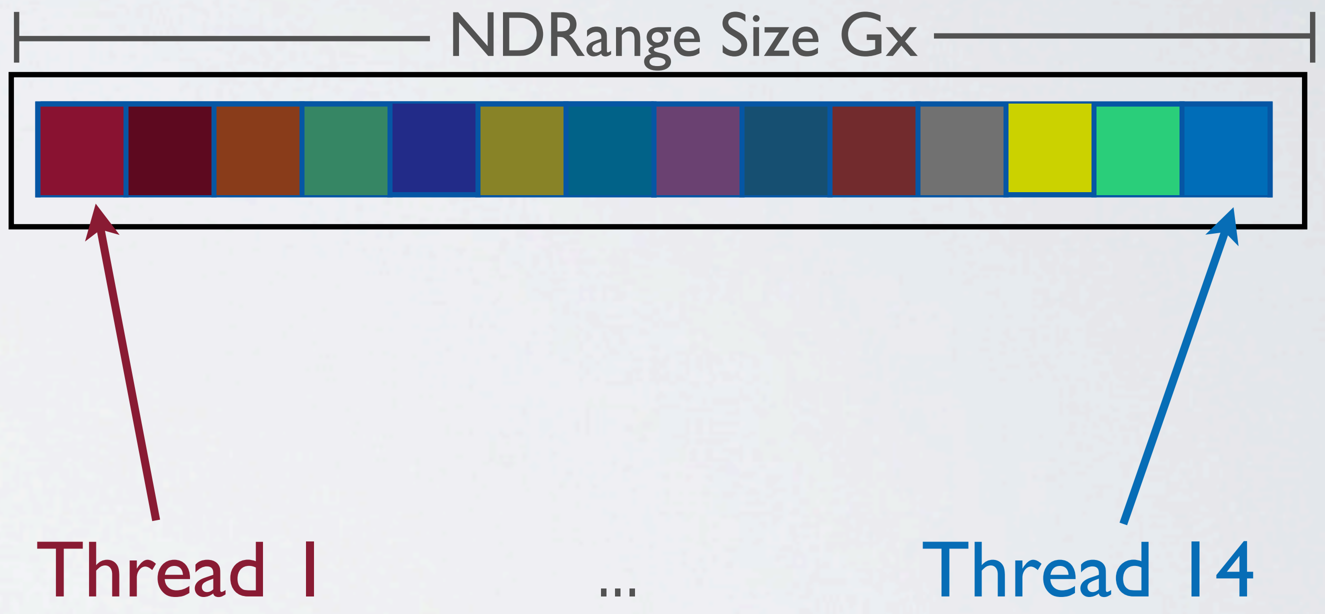
```
image2d_t, image3d_t
```

```
size_t get_local_id(uint dimindx);
```

```
float2 or cl_float2
```

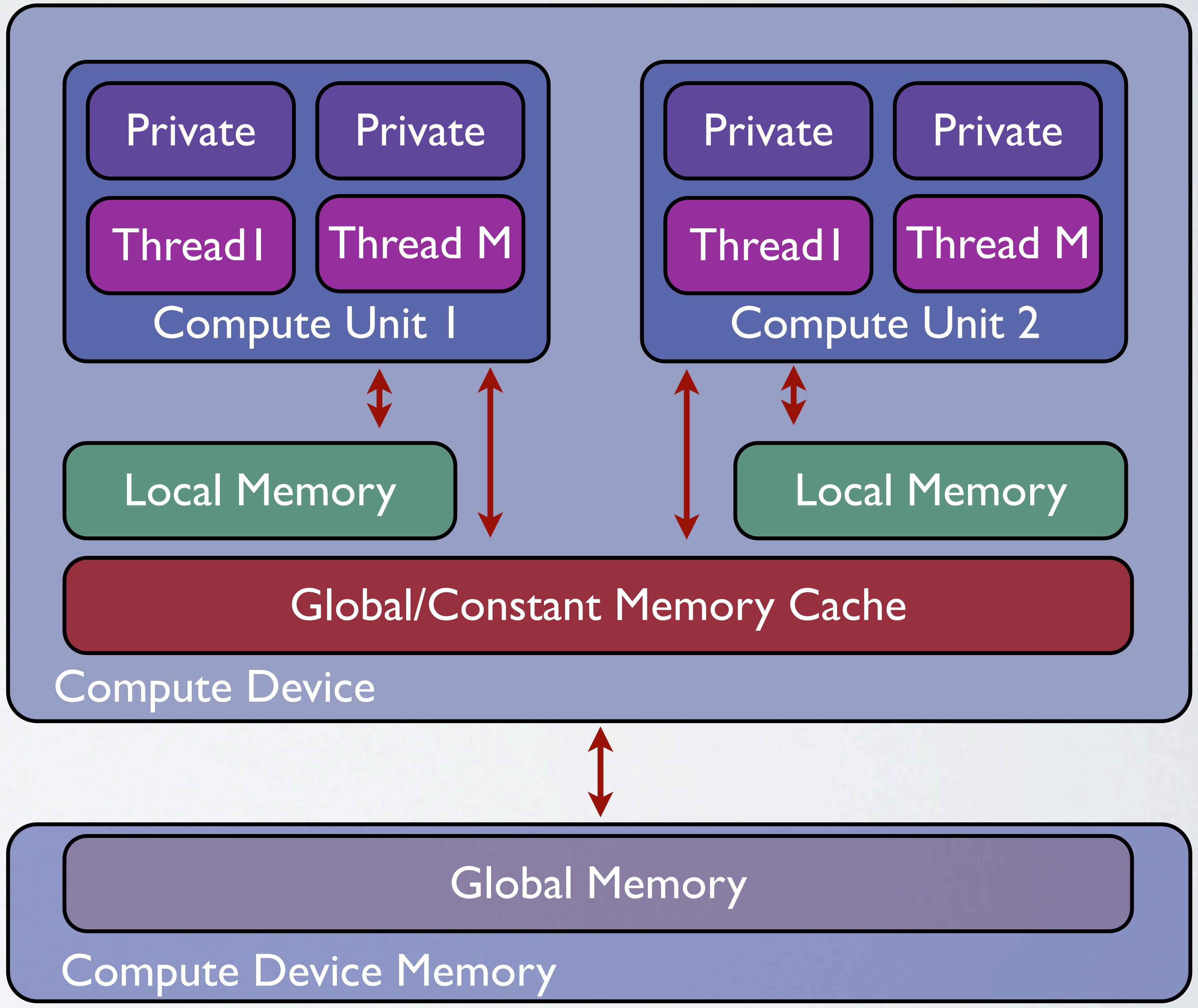
OPENCL KERNELS

- On the GPU each instance of a kernel executing (work-item) is run as its own thread
- The GPU can host thousands of threads
- Threads on the GPU are extremely lightweight and are managed in hardware



OPENCL ADDRESS SPACES

- There are four address spaces
  - __private (CUDA local)
  - __local (CUDA shared)
  - __constant (CUDA constant)
  - __global (CUDA global)



OPENCL API

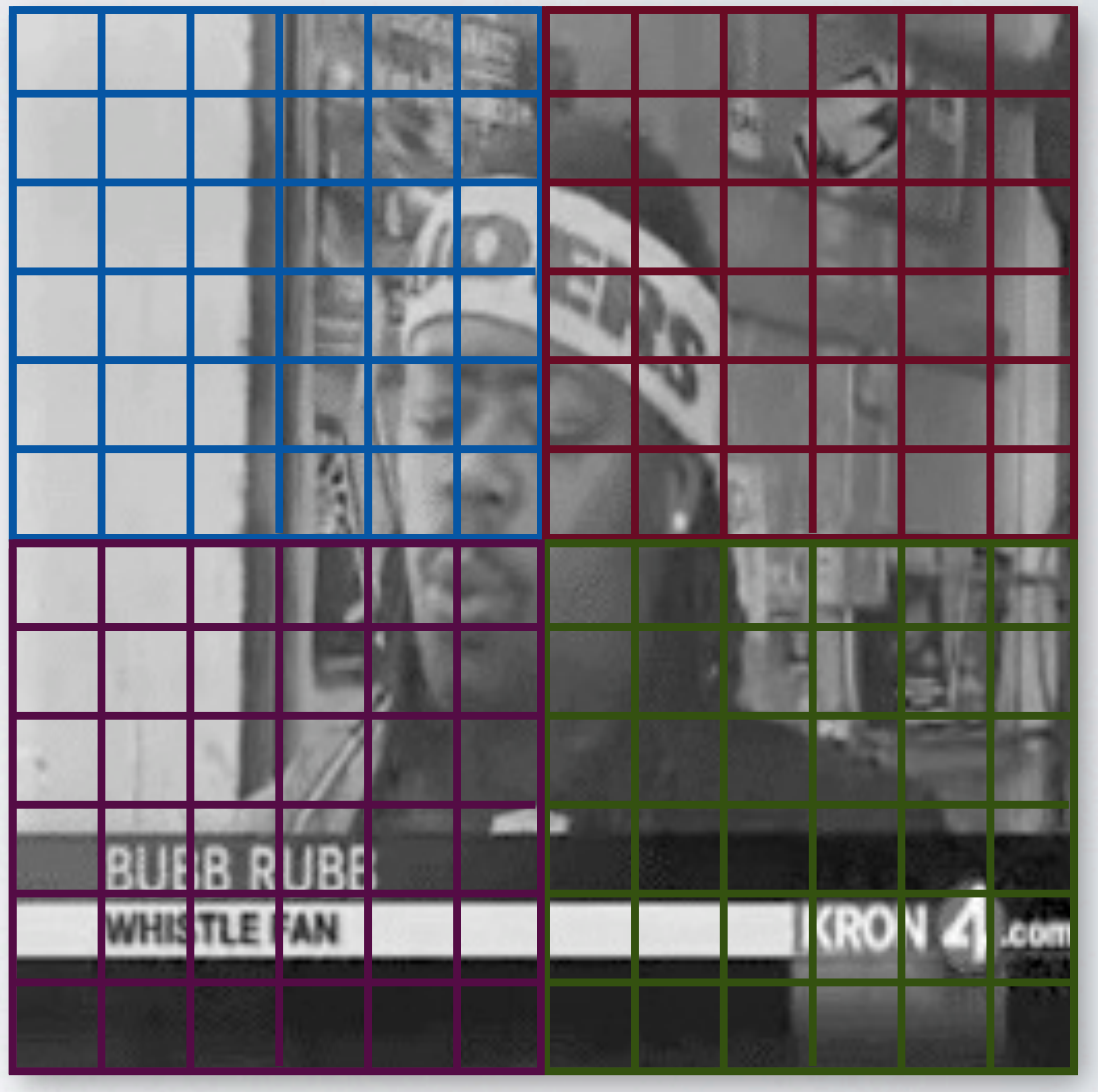
- The OpenCL API and specification can be viewed at http://www.khronos.org/opencl
- There are five main steps to run an OpenCL calculation
  - Initialization
  - Allocate resources
  - Creating programs/kernels
  - Execution
  - Tear down

EXAMPLE CALCULATION

- Process a 2D array of data on the GPU
- The data comes from (for example) an image file or other data source
- The details of calculation are not important for this example

EXAMPLE CALCULATION

- Process a 2D array of data on the GPU
- The data comes from (for example) an image file or other data source
- The details of calculation are not important for this example



INITIALIZATION

- Selecting a device and creating a context in which to run the calculation

```
cl_int err;
cl_context context;
cl_device_id devices;
cl_command_queue cmd_queue;

err = clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &devices, NULL);
context = clCreateContext(0, 1, &devices, NULL, NULL, &err);
cmd_queue = clCreateCommandQueue(context, devices, 0, NULL);
```

ALLOCATION

- Allocation of memory/storage that will be used on the device and push it to the device.

```
cl_mem ax_mem = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                          atom_buffer_size, NULL, NULL);

err = clEnqueueWriteBuffer(cmd_queue, ax_mem, CL_TRUE, 0,
                                    atom_buffer_size, (void*)ax, 0,NULL,NULL);
clFinish(cmd_queue);
```

PROGRAM/KERNEL CREATION

- Programs and kernels are read in from source and compiled or loaded as binary

```
cl_program program[1];
cl_kernel kernel[1];

program[0] = clCreateProgramWithSource(context,1,
                                (const char**)&program_source, NULL, &err);

err = clBuildProgram(program[0], 0, NULL, NULL, NULL, NULL);
kernel[0] = clCreateKernel(program[0], "mdh", &err);
```

EXECUTION

- Arguments to the kernel are set and the kernel is executed on all data

```
size_t global_work_size[2], local_work_size[2];
global_work_size[0] = nx; global_work_size[1] = ny;
local_work_size[0] = nx/2; local_work_size[1] = ny/2;

err  = clSetKernelArg(kernel[0],  0, sizeof(cl_mem), &ax_mem);

err = clEnqueueNDRangeKernel(cmd_queue, kernel[0], 2, NULL,
                              &global_work_size, &local_work_size,
                              0, NULL, NULL);
```

TEAR DOWN

- As part of the process we read back the results to the host and clean up memory

```
err = clEnqueueReadBuffer(cmd_queue, val_mem, CL_TRUE, 0,
                          grid_buffer_size, val, 0, NULL, NULL);

clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleaseContext(context);
```

MORE INFORMATION

- MacResearch.org
  - OpenCL - http://www.macresearch.org/opencl
  - Amazon Store - http://astore.amazon.com/macreseorg-20
- Khronos OpenCL - http://www.khronos.org/opencl
- Bubb Rubb on YouTube - http://bit.ly/r3ZF