GPU Computing

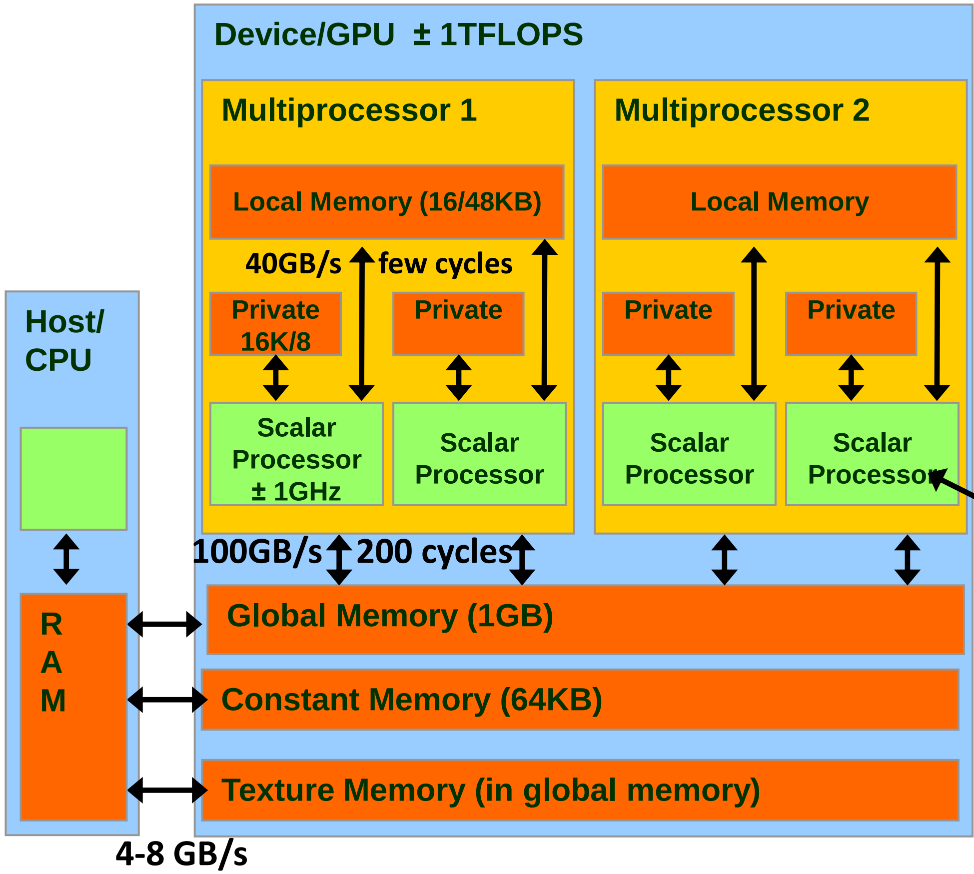
Lesson 2: Programming GPUs

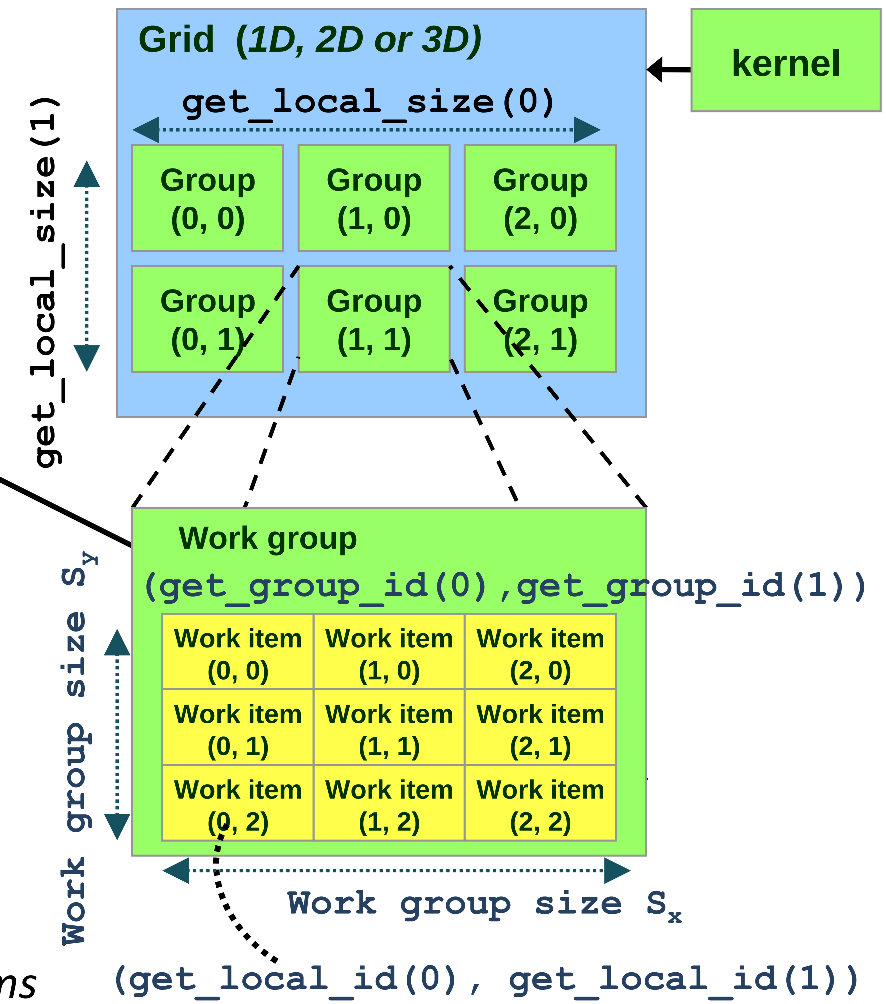
Jan Lemeire
2020-2021

Levels of Understanding

- Level 0
  - Host code
- Level 1
  - Parallel execution on the device
- Level 2
  - Device model and work groups
- Level 3 *=> explained in lesson 3*
  - Hardware threads & SIMT

GPU Concepts





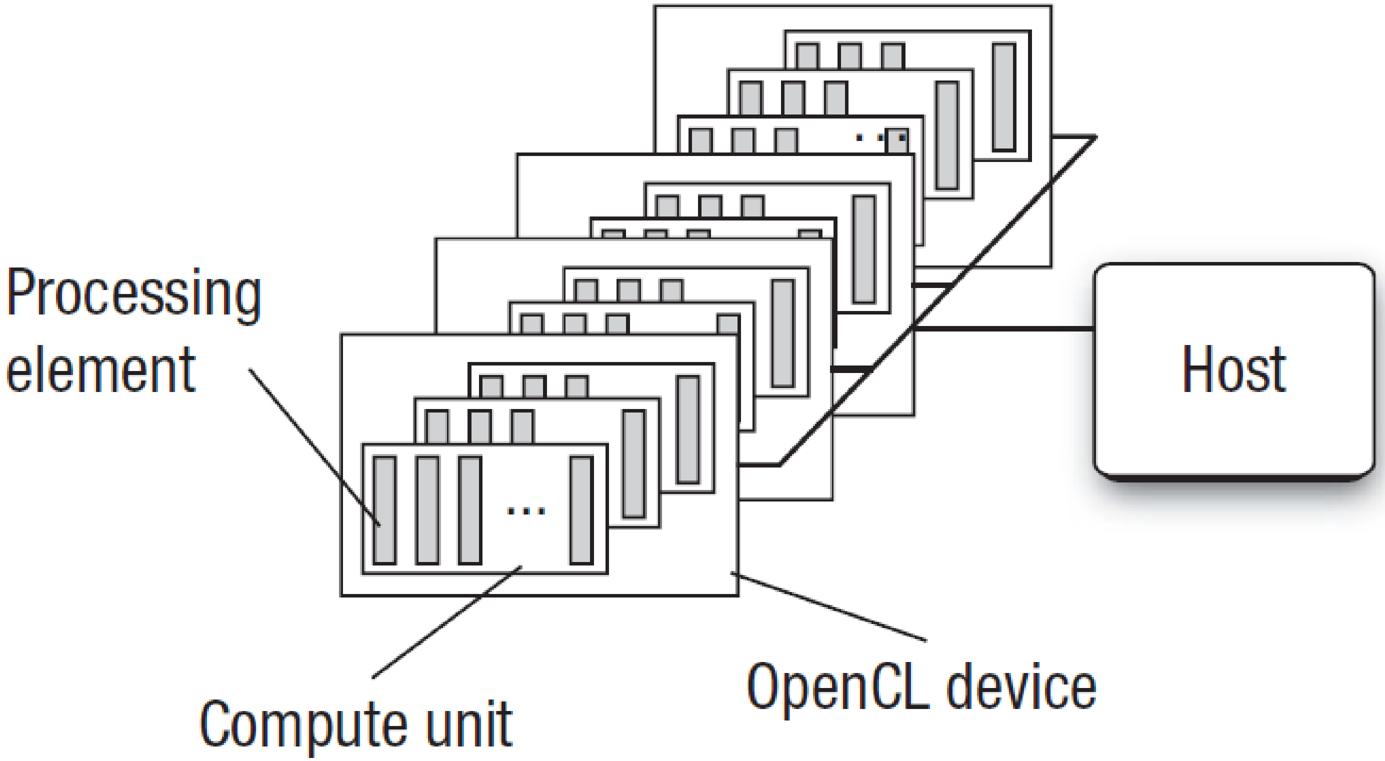
Max #work items per work group: 1024
Executed in warps/wavefronts of 32/64 work items
Max work groups simultaneously on MP: 8
Max active warps/wavefronts on MP: 24/48

OpenCL terminology

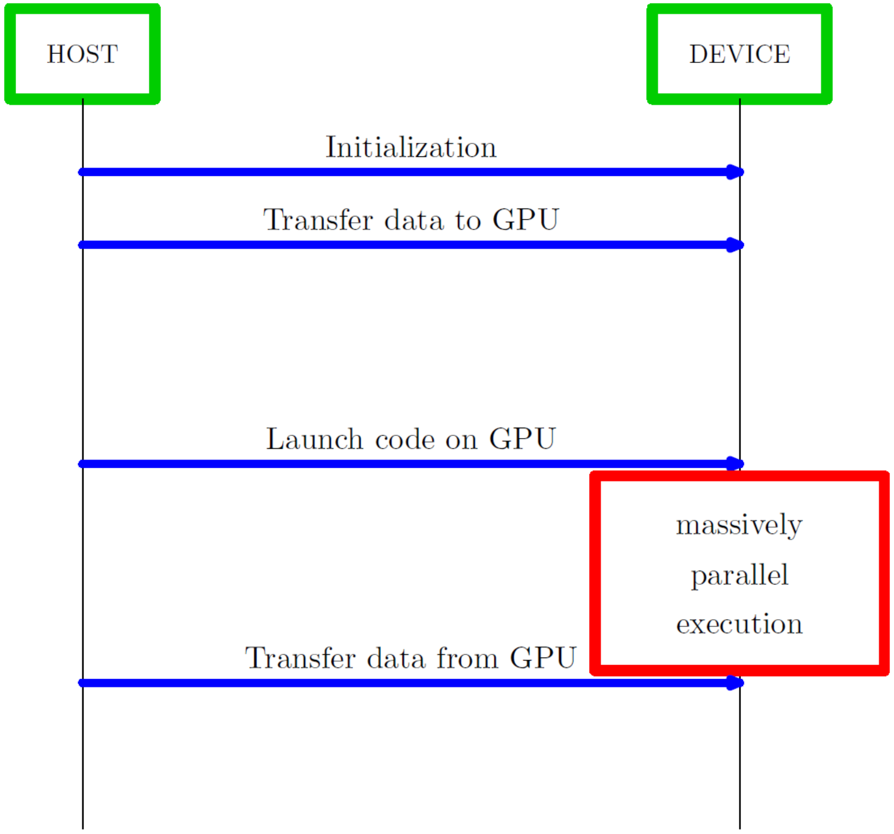
Level 0

Host Code

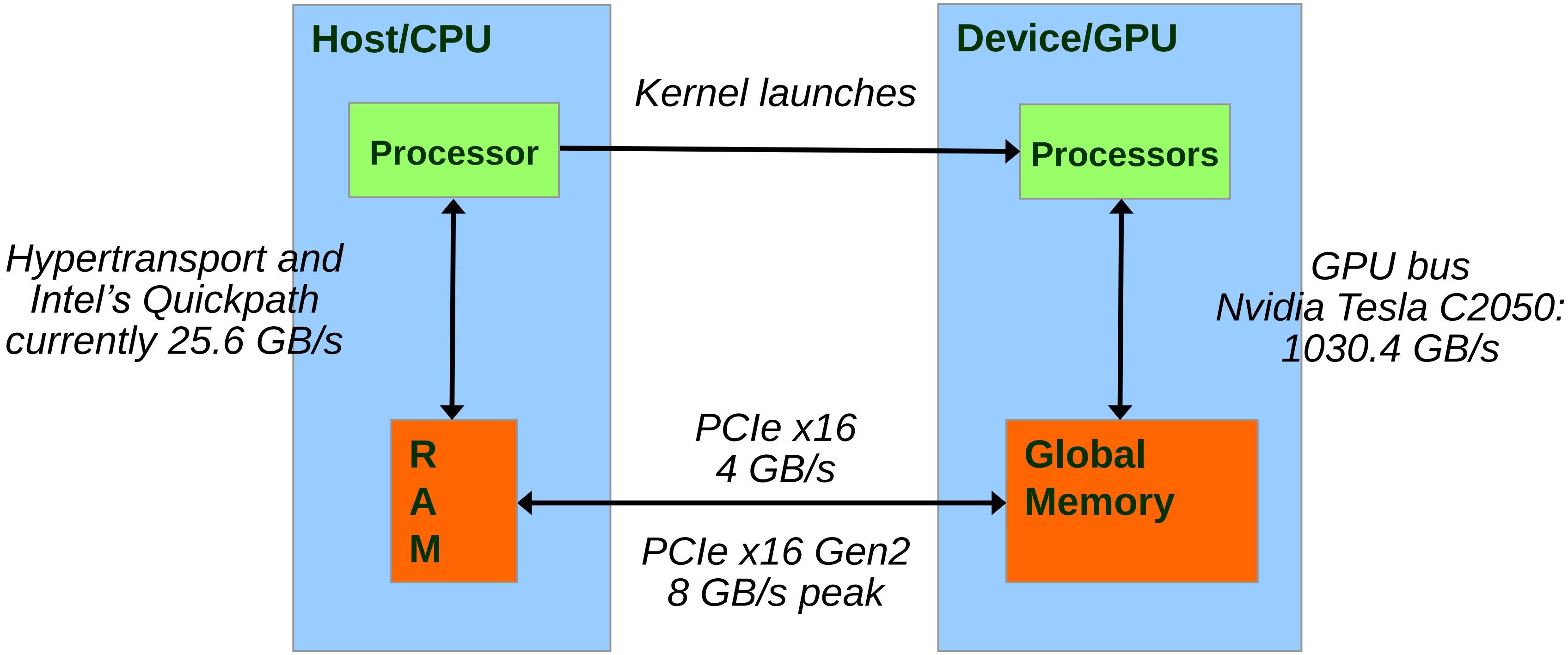
A Heterogeneous System
Host and Device



Typical Sequence of Events



Host (CPU) – Device (GPU)



OpenCL

- We need a way to
  - Modify our program to use accelerators
  - Specify the code that needs to run on the accelerators

- OpenCL
  - A host API
  - OpenCL C language
  - A model of
    - A heterogeneous system
    - An OpenCL device

- https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/

OpenCL Resources
A small sample

- www.khronos.org
- www.iwocl.org (*)
- www.streamcomputing.eu (*)
- developer.amd.com/tools-and-sdks/opencl-zone/
- www.eriksmistad.no/category/opencl/
- www.youtube.com
  - AJ Guillon

(*) These sites include references to books

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

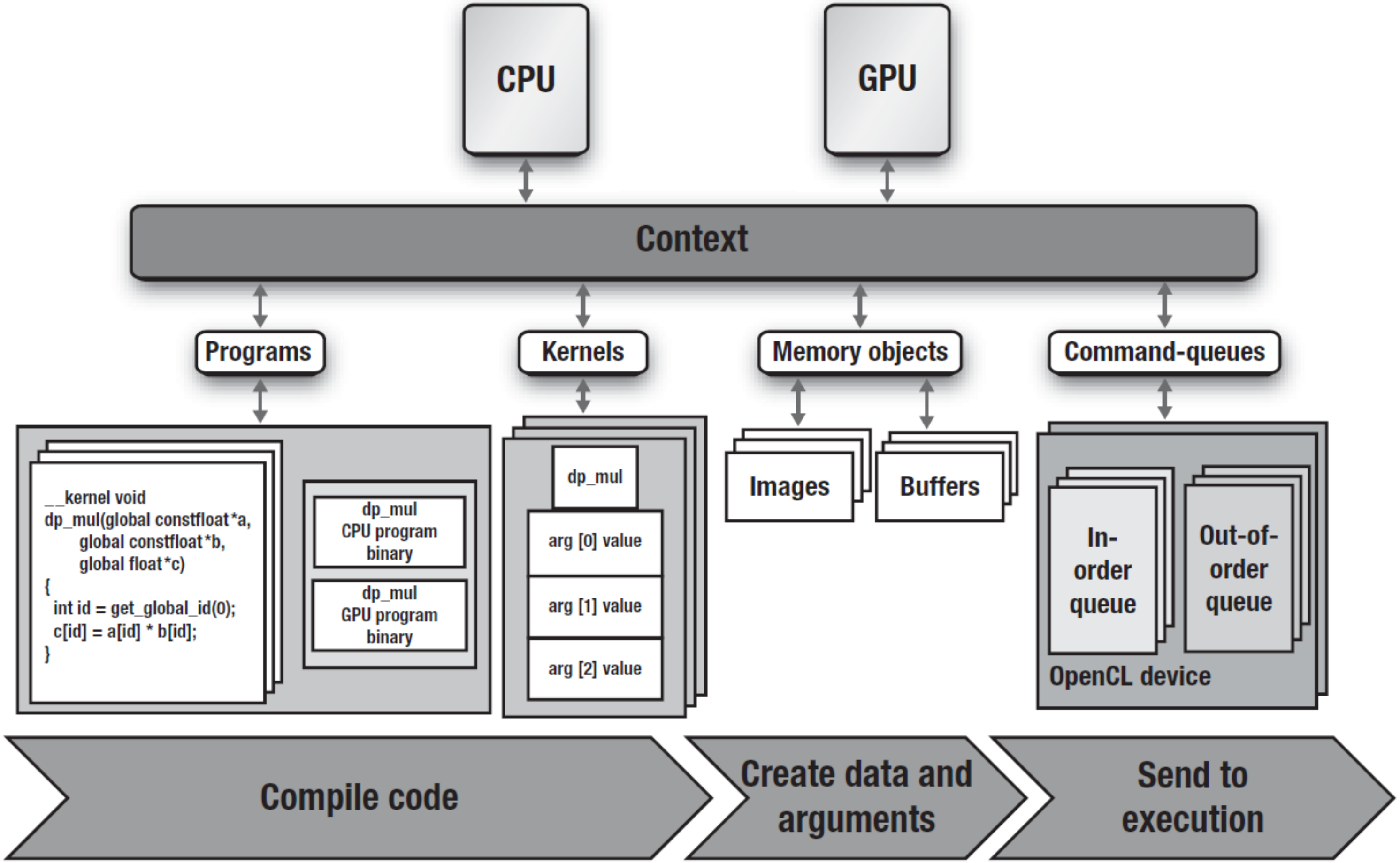
CUDA Working Group

HOST API

- We need only a little knowledge:
  1. Select the appropriate GPU.
  2. Allocate memory on the GPU.
  3. Transfer data between CPU and GPU.
  4. Compile and run code for/on the GPU.
- Understand what has to be modified.
- Seasoned programmers consult the manual pages

  https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml

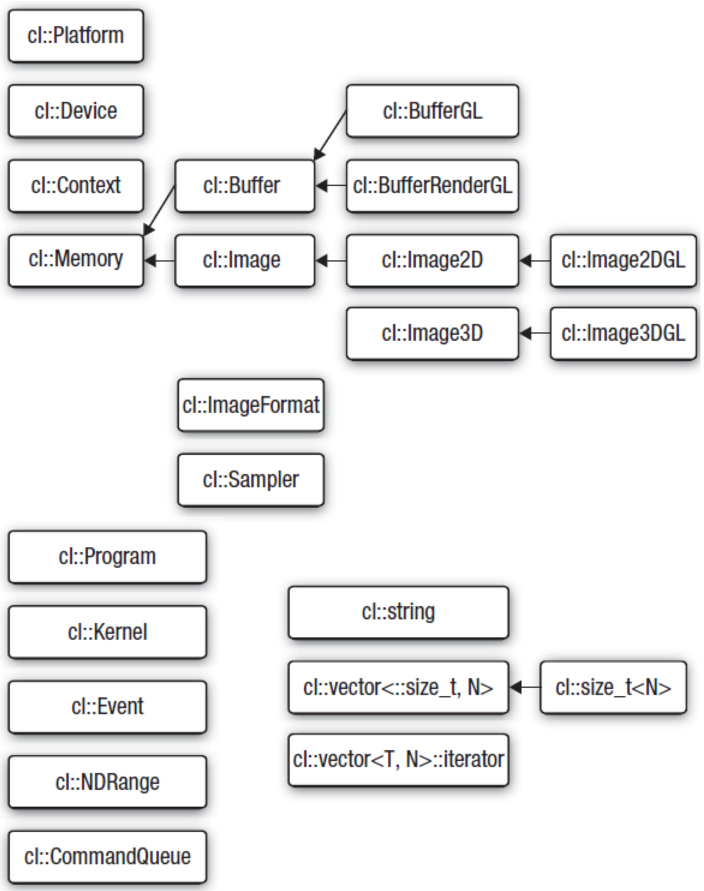
Host API


CPU
GPU
Context
Programs
Kernels
Memory objects
Command-queues
__kernel void
dp_mul(global constfloat*a,
global constfloat*b,
global float*c)
{
int id = get_global_id(0);
c[id] = a[id] * b[id];
}
dp_mul
CPU program
binary
dp_mul
GPU program
binary
dp_mul
arg [0] value
arg [1] value
arg [2] value
Images
Buffers
In-order queue
Out-of-order queue
OpenCL device
Compile code
Create data and arguments
Send to execution

Host API
Concepts

Platform	An OpenCL implementation e.g. AMD, Intel, NVIDIA, …
Device	An accelerator beloning to a platform
Context	A container object to deal with computation on the associated devices
Command Queue	Interface with a device. Used to send commands to the device
Program	A code container. Created from source or existing binaries
Kernel	A function to be run on a device
Buffer	A memory area on a device
NDRange	An execution configuration. See later

HOST API
C++ Wrapper



- Pros
  - Briefer
    - Exceptions instead of error handling
    - Certain methods equivalent to two C API function calls
  - Automatic cleanup
- Cons
  - May not be up to date
  - Man page mapping
    - Need some experience

HOST API
Technicalities

- Code for the exercises is provided
  - Includes the necessary header files
  - Includes a library file for Windows platforms
  - A VS solution is included

- Other systems may need more work
  - OS X: has OpenCL out of the box
  - Linux: you will need the appropriate library file

- A device driver that supports OpenCL
  - Compiles device code at runtime

The OpenCL Host API
OpenCL Hello World (1)

- Initializing OpenCL

```
std::vector<cl::Platform> platforms;
std::vector<cl::Device> devices;
cl::Platform::get(&platforms);
platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);

cl::Context context(devices);

cl::CommandQueue queue(context, devices[0], CL_QUEUE_PROFILING_ENABLE);
```

deviceQuery

on my machine…

```
Platform AMD Accelerated Parallel Processing
    Device Name:    Tahiti
    Device Type:    CL_DEVICE_TYPE_GPU
    OpenCL Version: OpenCL C 1.2
    Cache Type:     CL_READ_WRITE_CACHE
    Max work group size: 256
--------------------------------------------------
    Device Name:    Intel(R) Core(TM) i7 CPU         960  @ 3.20GHz
    Device Type:    CL_DEVICE_TYPE_CPU
    OpenCL Version: OpenCL C 1.2
    Cache Type:     CL_READ_WRITE_CACHE
    Max work group size: 1024
--------------------------------------------------
Platform NVIDIA CUDA
    Device Name:    Tesla C2050
    Device Type:    CL_DEVICE_TYPE_GPU
    OpenCL Version: OpenCL C 1.1
    Cache Type:     CL_READ_WRITE_CACHE
    Max work group size: 1024
--------------------------------------------------
    Device Name:    GeForce GTX 650 Ti
    Device Type:    CL_DEVICE_TYPE_GPU
    OpenCL Version: OpenCL C 1.2
    Cache Type:     CL_READ_WRITE_CACHE
    Max work group size: 1024
--------------------------------------------------
```

See project **deviceQuery**

The OpenCL Host API
OpenCL Hello World (2)

- Allocating memory
- Transferring data

```
unsigned int size = data_count*sizeof(cl_float);

cl::Buffer source_buf(context, CL_MEM_READ_ONLY, size);
cl::Buffer dest_buf(context, CL_MEM_WRITE_ONLY, size);

queue.enqueueWriteBuffer(source_buf, CL_TRUE, 0, size, source);

// ...

queue.enqueueReadBuffer(dest_buf, CL_TRUE, 0, size, dest);
```

See project **copyFloats**

The OpenCL Host API
OpenCL Hello World (3)

- Compiling and executing code

```
cl::Program program = jc::buildProgram(kernel_file, context, devices);
cl::Kernel kernel(program, kernel_name.c_str());

kernel.setArg<cl::Memory>(0, source_buf);
kernel.setArg<cl::Memory>(1, dest_buf);
kernel.setArg<cl_uint>(2, data_count);

cl::NDRange global(data_count); // number of work items

cl_ulong t = jc::runAndTimeKernel(kernel, queue, global) // nanoseconds
```

- *kernel_file*: name of text file containing OpenCL C code
- *kernel_name*: name of the kernel function
- Specify kernel arguments
- Specify number of work items (kernel threads)

Level 1

Parallel Execution on the Device

OpenCL C

A language based on C99

Extensions

- Function qualifiers

  `__kernel`

- Memory qualifiers

  `__global, __constant, __local, __private`

- Workspace query functions

  `get_global_id(dimidx), ...`

- Access qualifiers

  `__read_only, __write_only`

Limitations

- No recursion
- No function pointers
- No dynamic memory

OpenCL C

OpenCL Hello World (4)

- *kernel_file* contains a function called *floatCopy*
- *floatCopy* specifies the work of a single work item

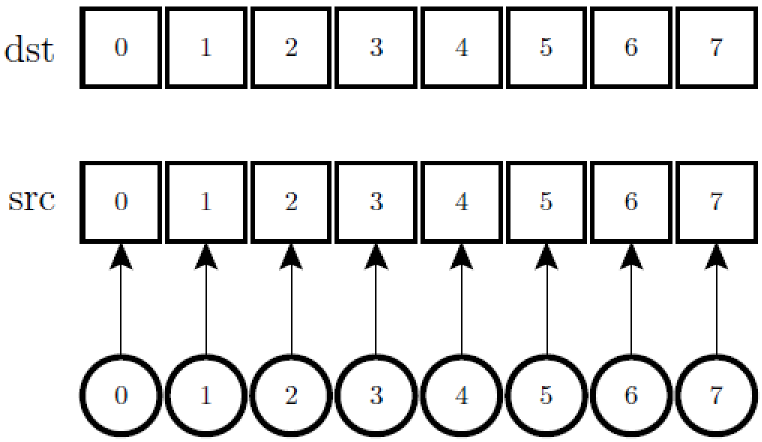
```
__kernel void floatCopy(
    __global float * source,
    __global float * dest,
    unsigned int     data_size
    )
{
    size_t index = get_global_id(0);

    if (index >= data_size) {
        return;
    }

    dest[index] = source[index];

    return;
}
```

Mapping of work items on data



OpenCL C

OpenCL Hello World (5)

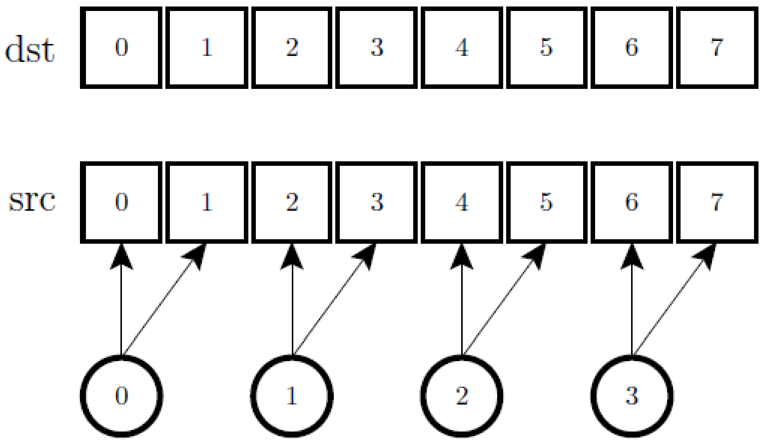
- The programmer specifies the number of work items
- Enough work items to handle all data items

```
__kernel void floatCopy(
    __global float * source,
    __global float * dest,
    unsigned int     data_size
    )
{
    size_t index1 = 2*get_global_id(0);
    size_t index2 = index1 + 1;

    dest[index1] = source[index1];
    if (index2 < data_size) {
        dest[index2] = source[index1];
    }

    return;
}
```

Mapping of work items on data



Second Example

- Implement a vector addition
  - Assume three lists A, B and C
  - Element i of C:
    - $C_i = A_i + B_i$;
- Extension:
  - One work item processes more than one data item

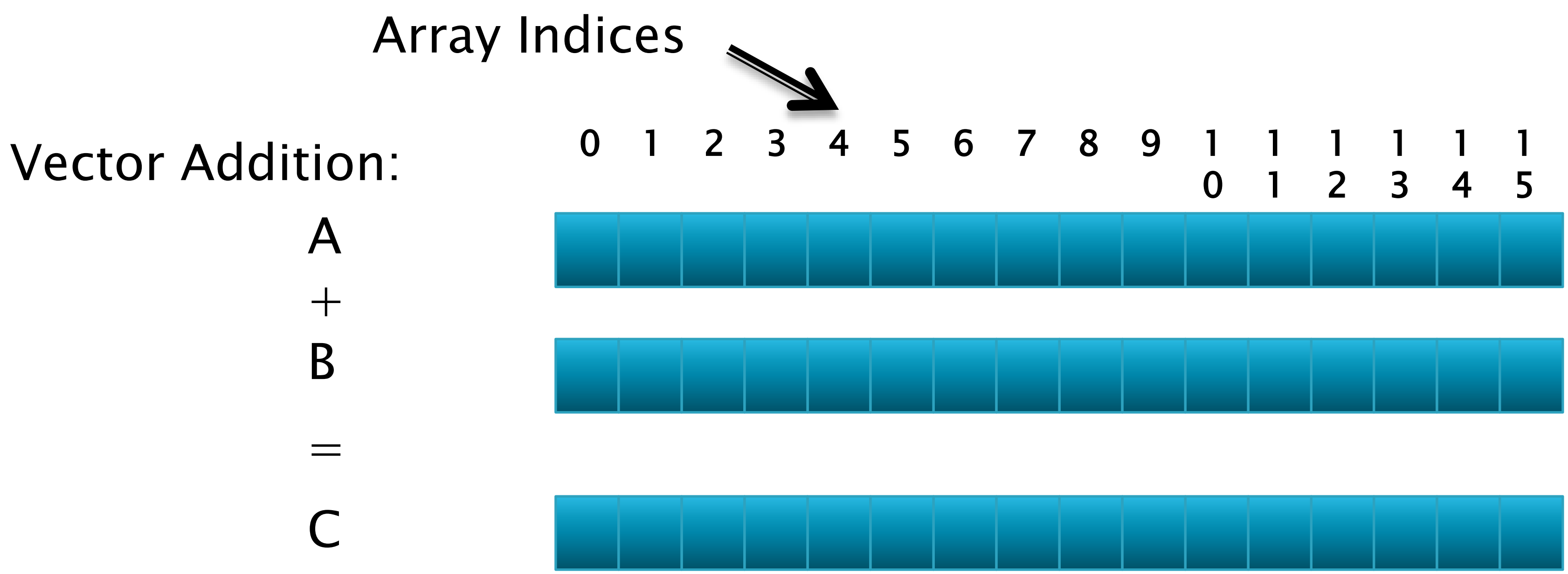
See project **sumInts**

Work item executes kernel

- Massively parallel programs are usually written so that each kernel thread (work item) computes one part of a problem
  - For vector addition, we will add corresponding elements from two arrays, so each thread will perform one addition
  - If we think about the thread structure visually, the threads will usually be arranged in the same shape as the data

Vector addition

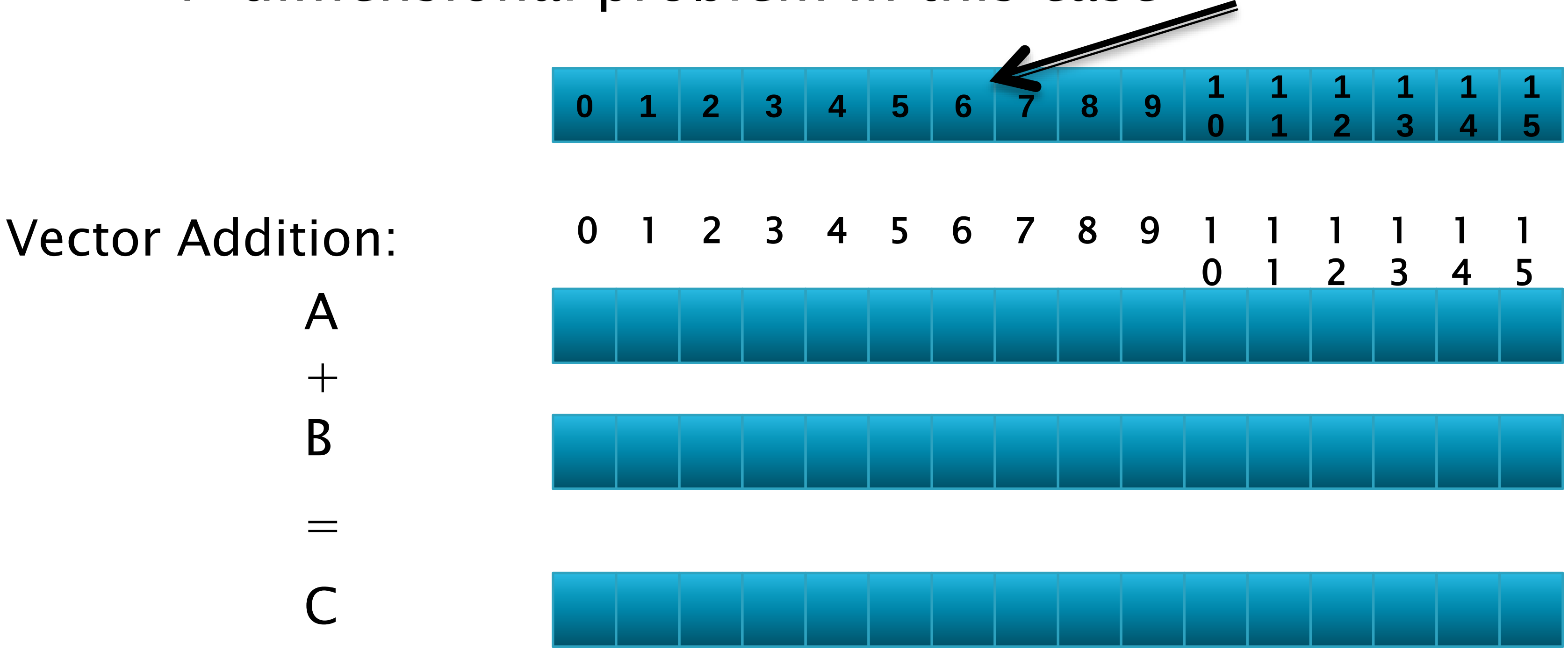
- Consider a simple vector addition of 16 elements
  - 2 input buffers (A, B) and 1 output buffer (C) are required



Vector addition

- Create thread structure (work items) to match the problem
  - 1-dimensional problem in this case

Work item IDs



Vector addition

- Each work item is responsible for adding the indices corresponding to its ID

Vector Addition:

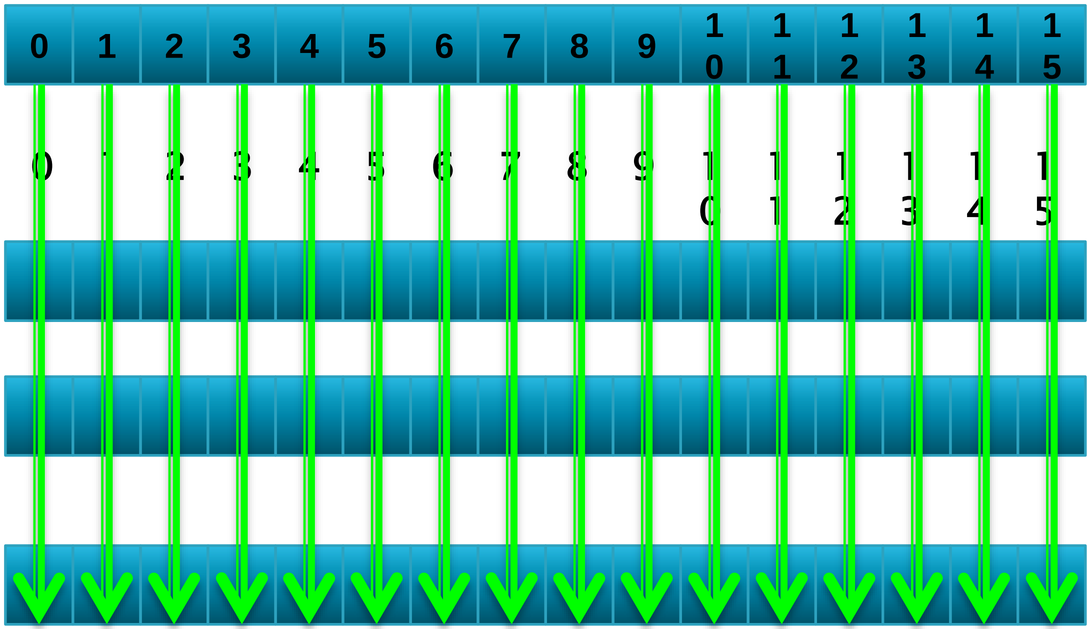
A

+

B

=

C



A work item is executed by a kernel thread

OpenCL Kernel code

```
__kernel void vectorAdd(__global const float * a,
__global const float * b, __global float * c)
{
  // Vector element index
  int nIndex = get_global_id(0);
  // addition
  c[nIndex] = a[nIndex] + b[nIndex];
}
```

- OpenCL kernel functions are declared using "__kernel".
- __global refers to global memory
- get_global_id(0) returns the ID of the thread in execution

Runtime math library

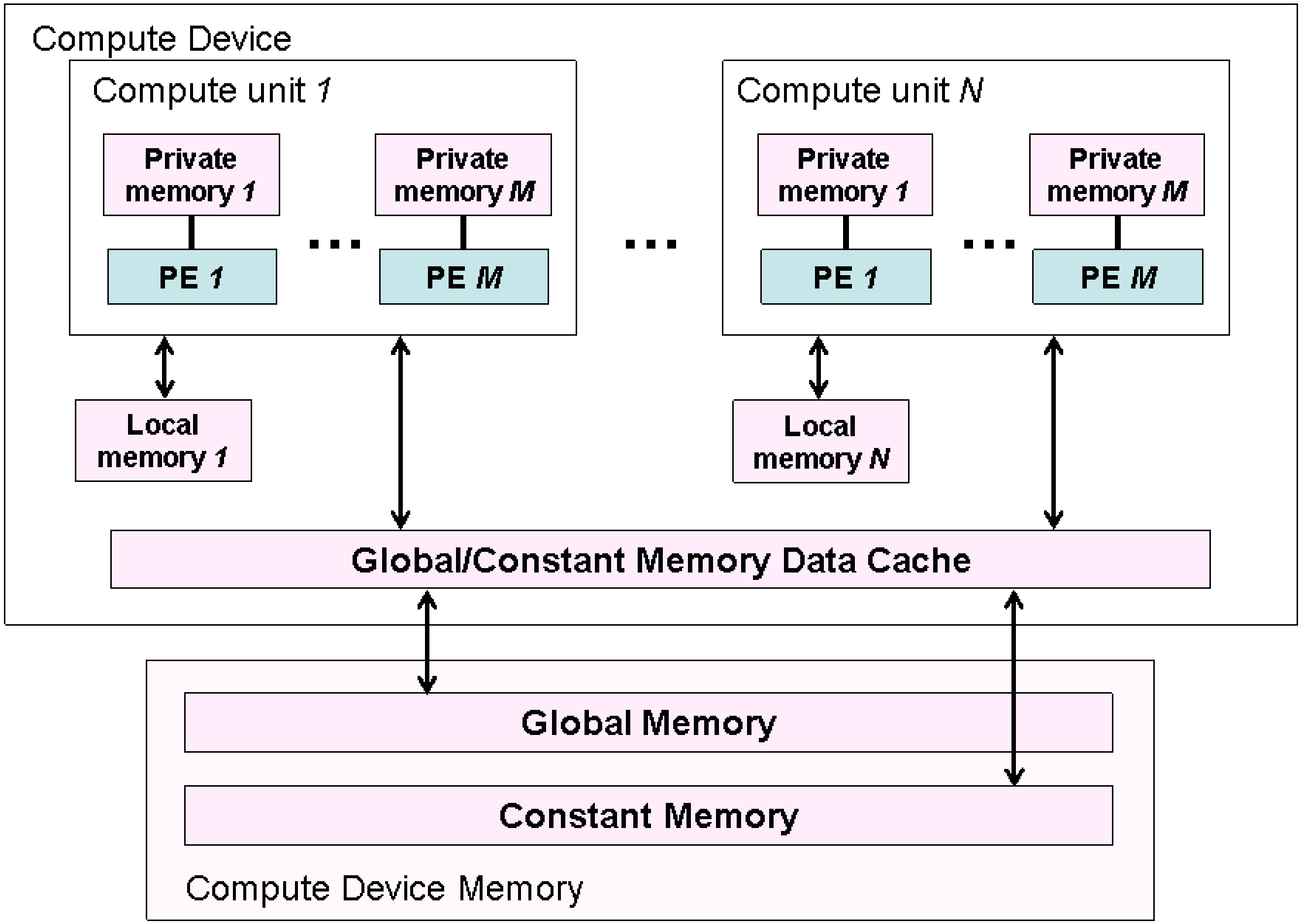
- Two ways to compute standard mathematical functions
  - func(): slow but precise
  - native_func(): less precise but fast
- For example
  - cos(), native_cos()
  - sqrt(), native_sqrt()
- Special hardware for native functions

Level 2

Device Model and Work Groups

OpenCL Device Model

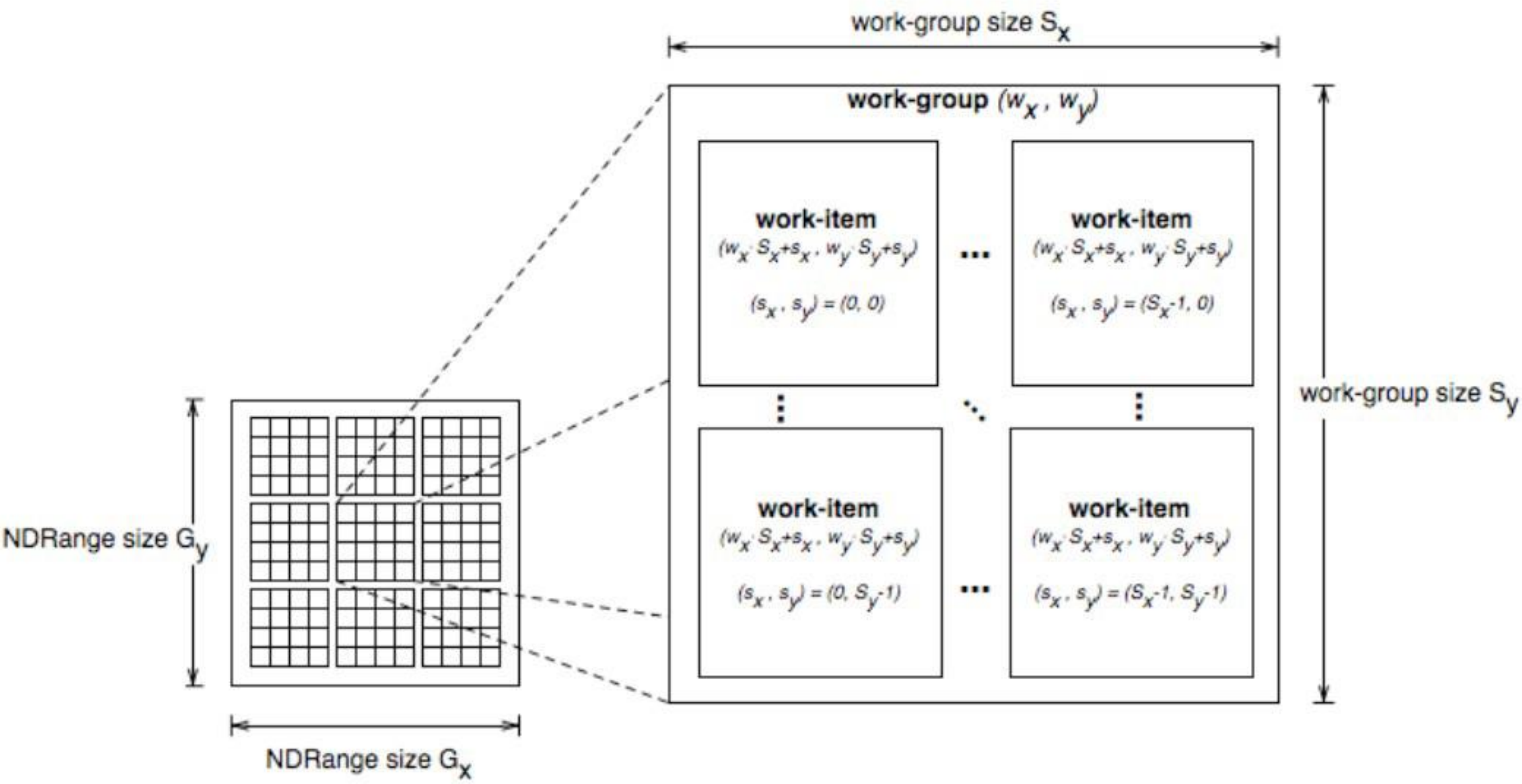
How do we exploit this?



Work groups

- Work items are divided into work groups
- A work group is executed on one compute unit
  - From start to end
- Work items of the same work group can share local memory
  - Kind of explicit cache
- Within a work group synchronization is possible
  - With the barrier statement.
- Work group size is determined by the programmer
  - As **local range** (local index space)
  - One size for all work groups

Architecture – Execution Model



OpenCL Work Space

Terminology and query functions

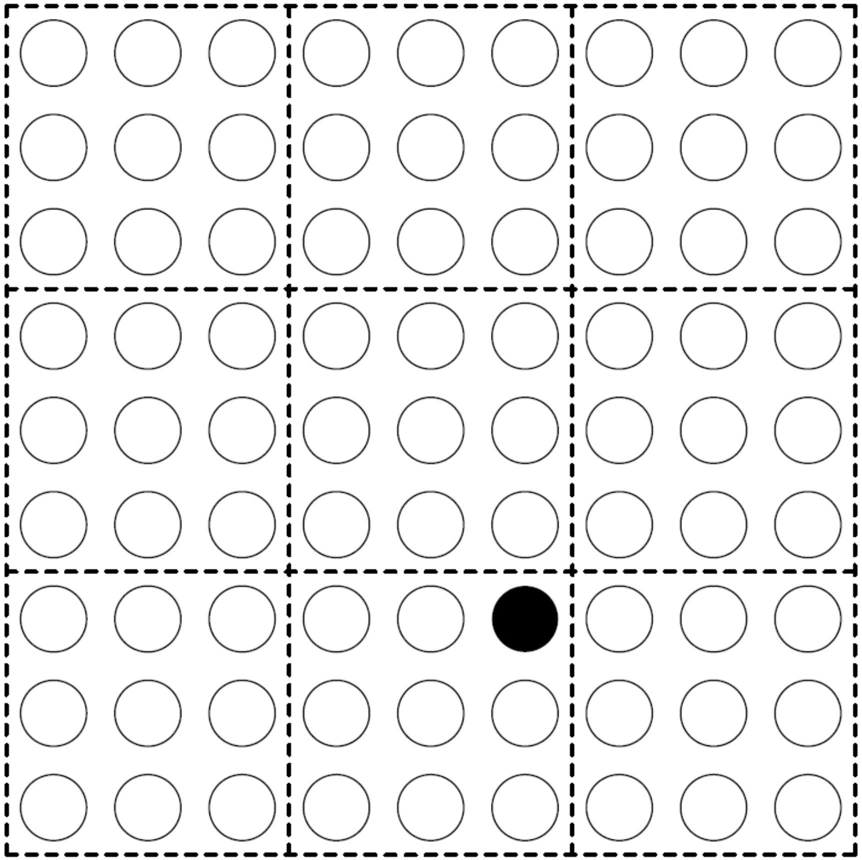
- N-dimensional range
  - index space
  - 1-, 2- or 3-dimensional
- Global NDRange: configuration of ALL work items
- Local NDRange: configuration of a work group
- Note:
  - Global and Local ranges must have the same number of dimensions!
  - Work group size in a certain dimension must be a whole divisor of the global size in this direction

- Query functions

```
get_global_id(dimidx)
get_global_size(dimidx)
get_group_id(dimidx)
get_local_id(dimidx)
get_local_size(dimidx)
get_num_groups(dimidx)
get_work_dim()
```

OpenCL Work Space

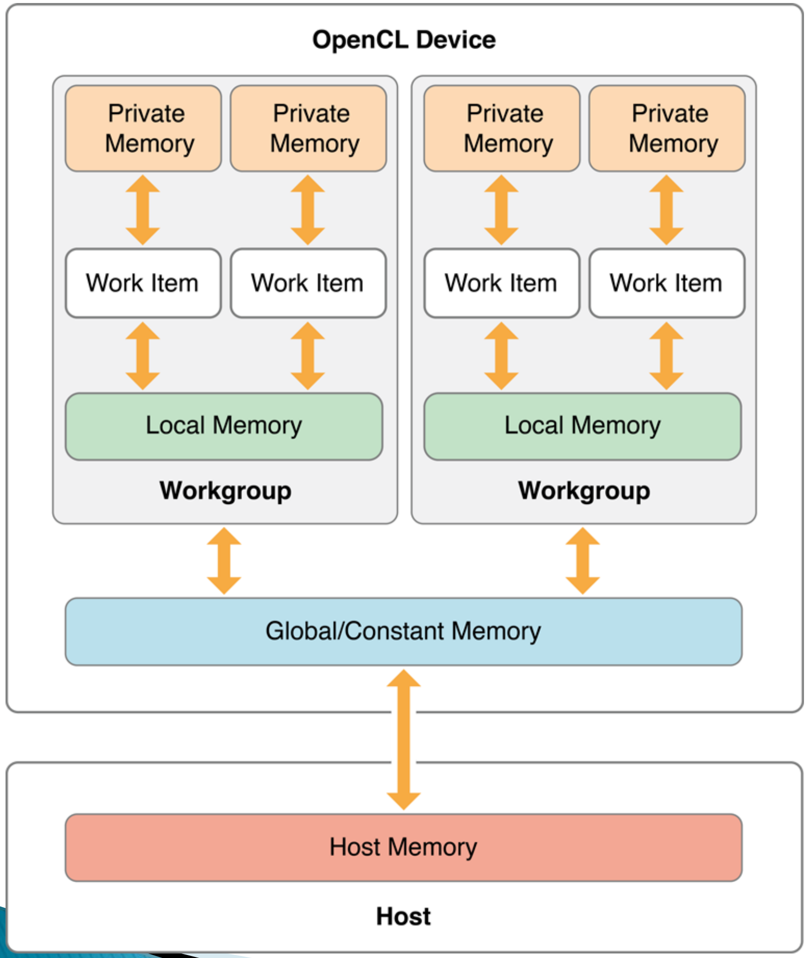
Quick test



```
get_global_id(0) = ______
get_global_id(1) = ______
get_global_size(0) = ______
get_global_size(1) = ______
get_group_id(0) = ______
get_group_id(1) = ______
get_local_id(0) = ______
get_local_id(1) = ______
get_local_size(0) = ______
get_local_size(1) = ______
get_num_groups(0) = ______
get_num_groups(1) = ______
get_work_dim() = ______
```

OpenCL Memory Model

Explicit Memory Hierarchy



- In your kernel code:

```
__kernel void similarity_constant_local
(
    __global float    * tags_min,          //  0
    __global float    * tags_max,          //  1
    __constant float  * query_min,         //  2
    __constant float  * query_max,         //  3
    __global float    * shifted_weights,   //  4
    __global float    * scores,            //  5
    __global uint     * indices,           //  6
    __global int      * offsets,           //  7
     unsigned int    tags_size,            //  8
     unsigned int    num_windows,          //  9
     unsigned int    index_resolution,     // 10
    __local float     * l_scores,          // 11
    __local uint      * l_indices,         // 12
    __local float     * l_tags_min,        // 13
    __local float     * l_tags_max,        // 14
    unsigned int        tag_count          // 15
)
{
    // kernel body omitted
}
```

OpenCL Memory Model

using local memory

(1) In the kernel body

```
#define N 256

__kernel void similarity_constant_local
(
    __global float * in,
    __global float * out
    unsigned int   size
)
{
    unsigned int index = get_global_id(0);
    __local float shared[N]; // constant
    // populate
    shared[get_local_id(0)] =
        index < size ? In[index] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);
    // use local memory
    // …
}
```

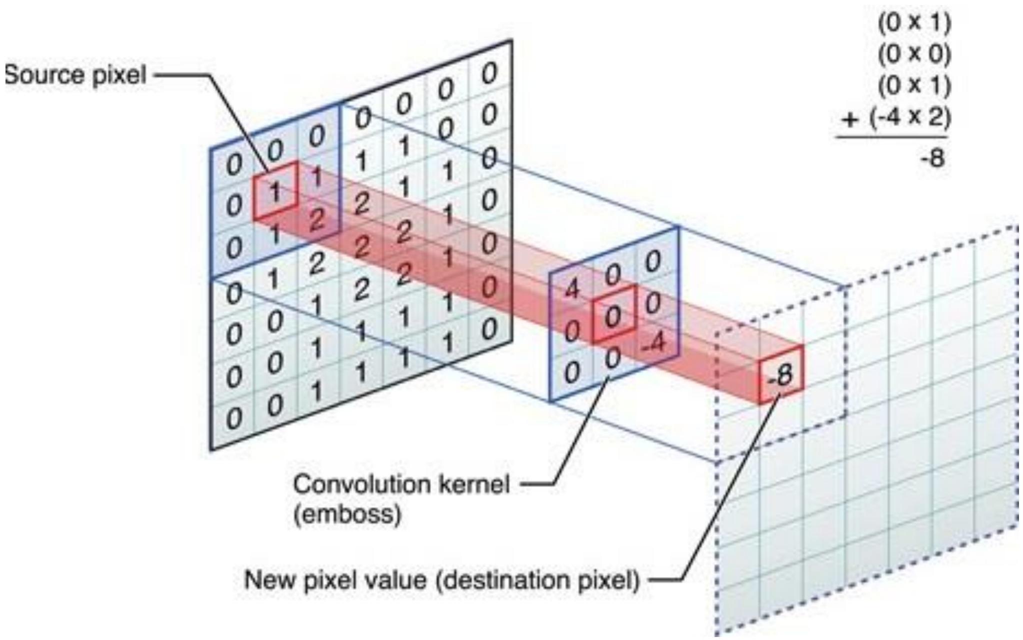
(2) As a kernel argument

```
__kernel void similarity_constant_local
(
    __global float * in,
    __global float * out,
    __local  float * shared,
    unsigned int   size
)
{
    unsigned int index = get_global_id(0);

    // populate
    shared[get_local_id(0)] =
        index < size ? In[index] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);
    // use local memory
    // …
}
```

```
kernel.setArg<cl::LocalSpaceArg>(2, cl::__local(N)); // N can be variable
```

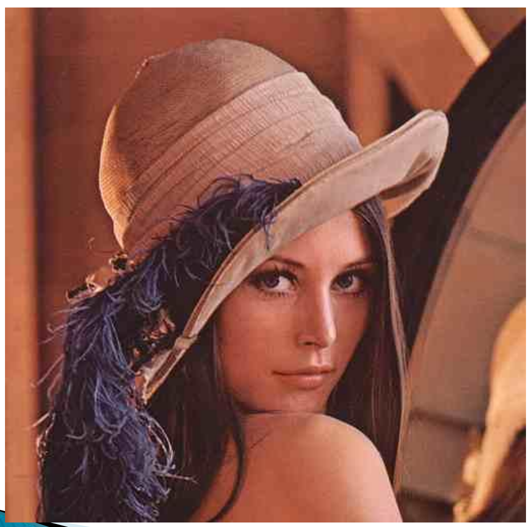
Example: convolution



Parallelism: +++
Locality: ++
Work/pixel: ++

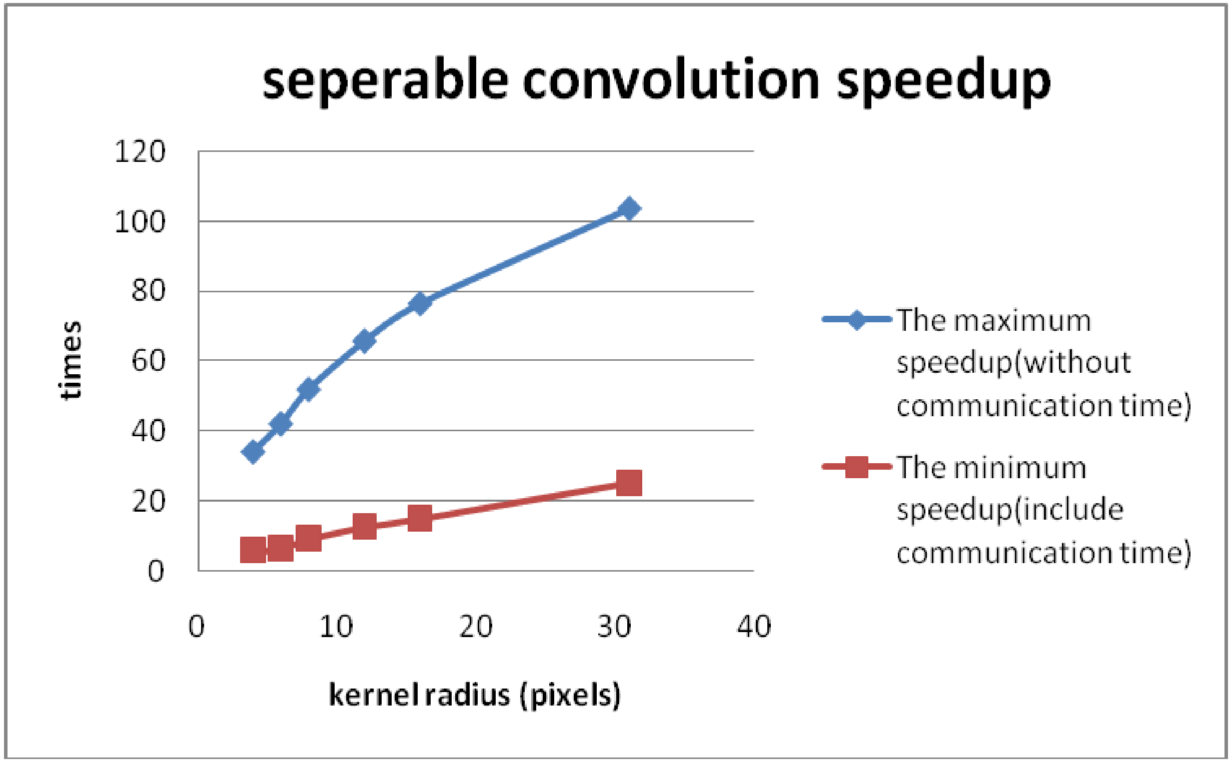
3x3 kernel (also called *filter* or *mask*) is applied to each pixel of the image

Examples of convolution

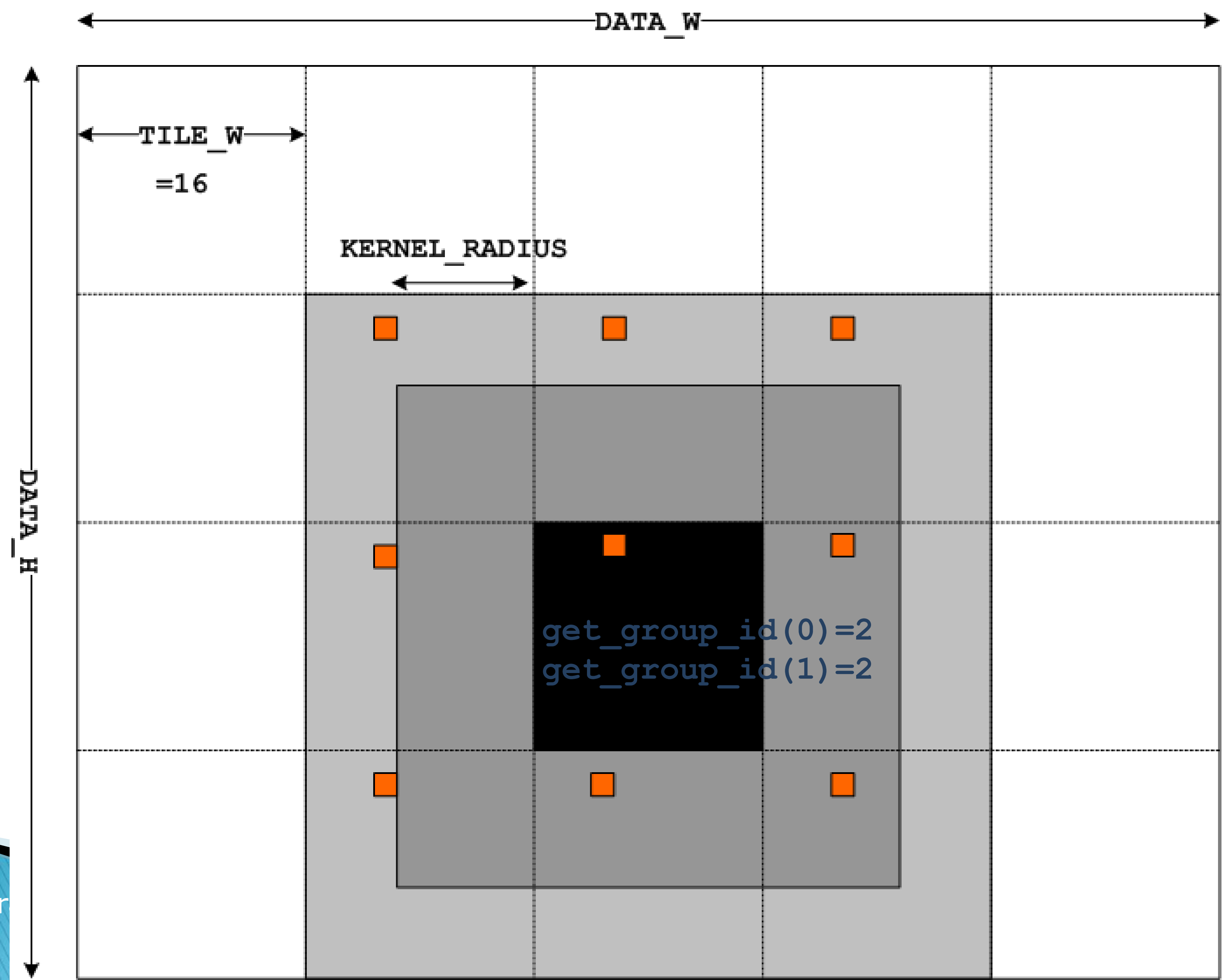


Edge detection with sobel filter

Speedup


seperable convolution speedup
times
120
100
80
60
40
20
0
0
10
20
30
40
kernel radius (pixels)
The maximum speedup(without communication time)
The minimum speedup(include communication time)

Convolution on GPU



Convolution Kernel Code

```
__kernel void convolutionUsingSharedMemory(
      __global int *in, __global int *out, __local  int *in_local, __constant int *filter, int
filter_height, int filter_width)
{
   uint row = get_global_id(1);
   uint col = get_global_id(0);

   in_local[get_local_id(1) * get_local_size(0) + get_local_id(0)] =
      in[row * get_global_size(0) + col];
   … // copy 9 pixels to local

   barrier(CLK_LOCAL_MEM_FENCE);
   int sum=0;
   for (int i = 0; i< filter_height; ++i)
      for (int j = 0; j< filter_width; ++j)
           sum += filter[…] * in_local[…];

   out[row * get_global_size(0) + col] = sum;
}
```

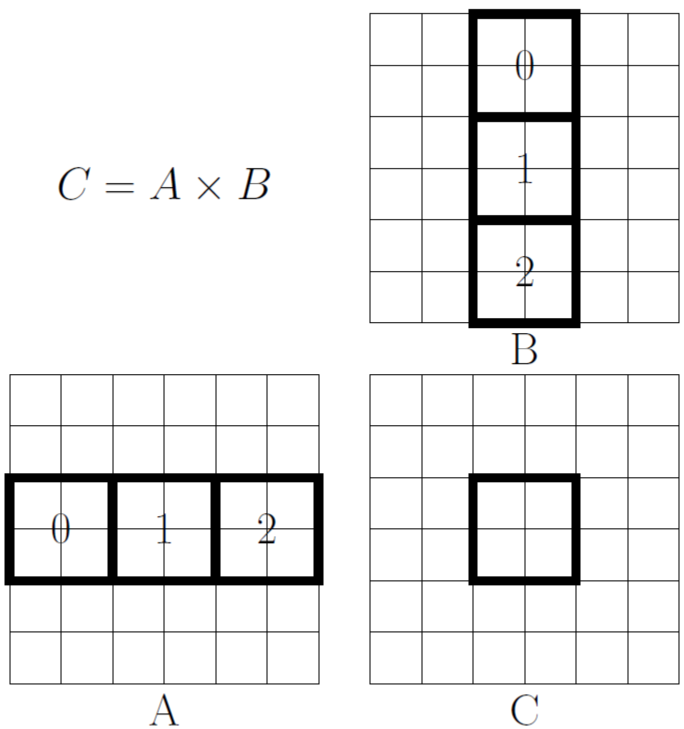
OpenCL Memory Model

using local memory – example

Matrix Multiplication

Happens in 3 iterations: first blocks 0 are multiplied, then 1 are multiplied and added, and at last blocks 2

$$C = A \times B$$



Device code

```
__kernel void mul(__global int *A, __global int *B, __global int *C,
int size) {
    __local int sharedA[16][16];
    __local int sharedB[16][16];

    int sum    = 0;
    int aStart = get_global_id(1)*size + get_local_id(0);
    int aEnd   = aStart + size;
    int bStart = get_local_id(1)*size + get_global_id(0);
    int aStep  = 16;      // move 16 colums
    int bStep  = 16*size; // move 16 rows

    for (int a = aStart, b = bStart; a < aEnd; a += aStep, b += bStep){
        sharedA[get_local_id(1)][get_local_id(0)] = A[a];
        sharedB[get_local_id(1)][get_local_id(0)] = B[b];
        barrier(CLK_LOCAL_MEM_FENCE);
        #pragma unroll
        for (int j = 0; j < 16; ++j)
            sum += sharedA[get_local_id(0)][j] *
                   sharedB[j][get_local_id(0)];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    C[y*size + x] = sum;
}
```

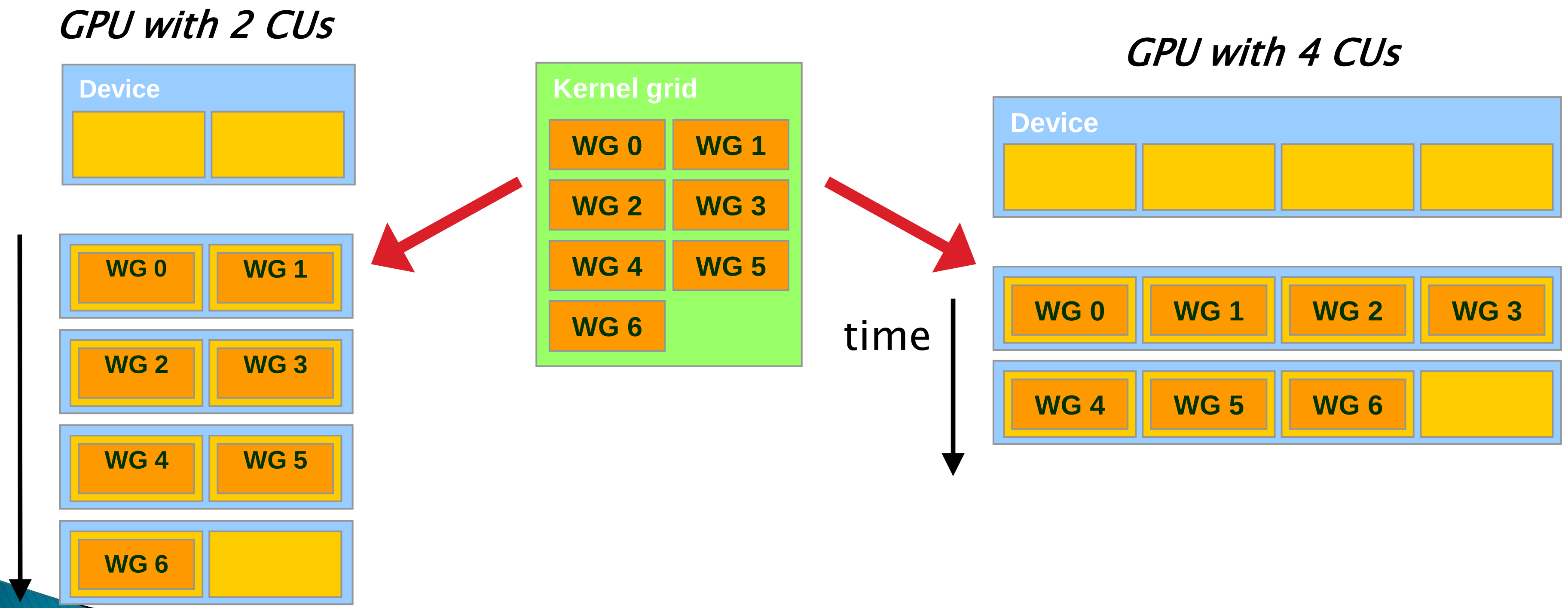
OpenCL Execution Model

- Execution of N work groups of m work items
- Work groups are assigned to **Compute Units (CUs)**
  - A work group stays there until it completes
- Compute units may execute multiple work groups concurrently
  - See later
- Work groups not yet assigned to a compute unit must wait
- The order in which work groups execute is non-deterministic

- Consequences
  - There can be no interaction between work groups
  - OpenCL code scales inherently

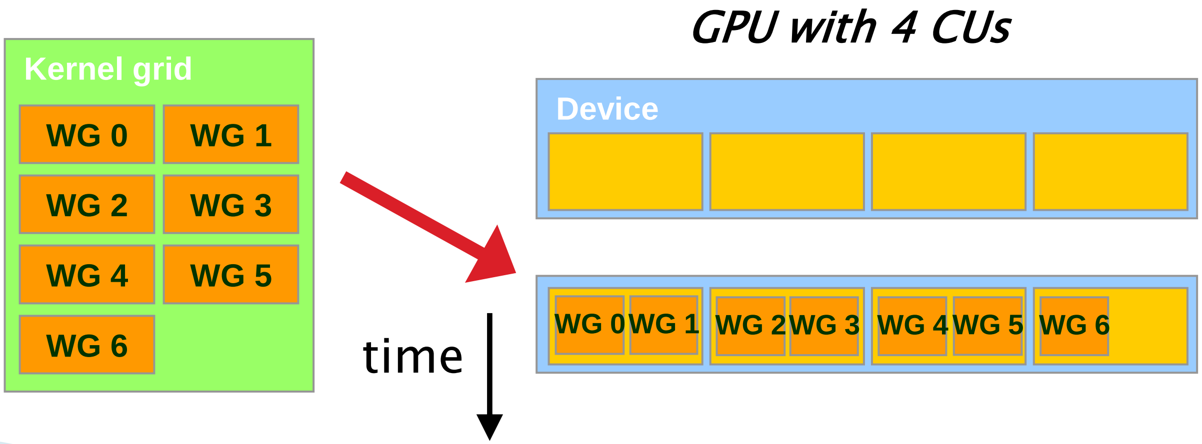
Work group execution

- Simple scheduler
  - Assigns work groups to available Compute Units (CUs)
  - Basically, a waiting queue for work groups
- Work groups (WGs) execute independently
  - Global Synchronization among work groups is not possible!



Multiple WGs per CU

- One CU can execute work groups concurrently
- Determined by available resources (hardware limits):
  - *Max. work groups simultaneously on CU: 8*
  - *Max. work items simultaneously on CU: 1024*
  - *Private memory (registers) per CU: 16/48KB*
  - *Local (shared) memory per CU: 16/32KB*



Exercise: Matrix Vector Operation

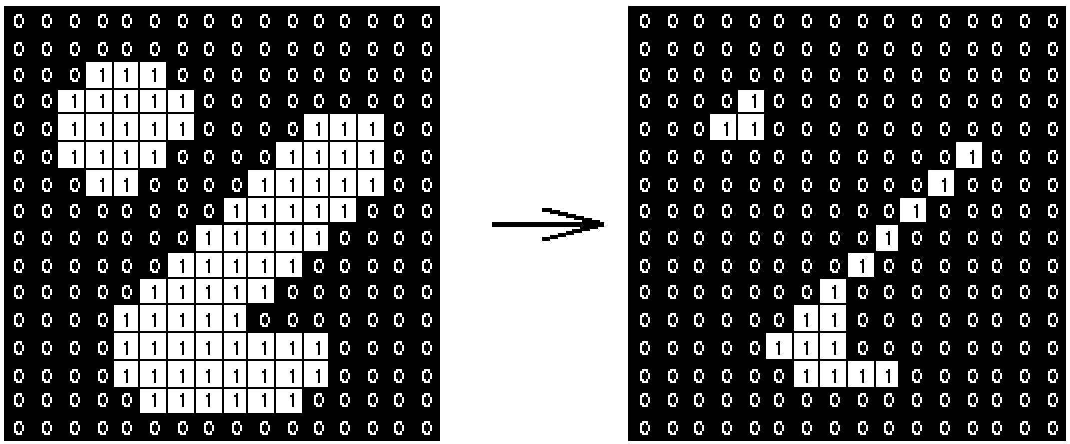
- Matrix A mxn
- Vector B n

- Computation?
  - Repeat N times:
    - A[i,j] = A[i,j] + A[i,j]*B[j]

- Observe
  - Data throughput in function of N
  - Computational throughput in function of N

Exercise: Erosion

- Typical operation in image processing
- Given an input pixel the value of the corresponding output pixel is the minimum of values of pixels under a mask centered on the input pixel
- Example Erosion with a 3x3 mask on a binary image:



- Implement erosion for one-dimensional data for a parameterizable mask width
  1. Doing everything in global memory
  2. Using local memory
- Try two-dimensional erosion

Images and OpenGL (optional info)

OpenCL Images
Background

- GPUs have texture memory
  - Special hardware to deal with images
  - Take advantage of:
    - 2D- caching
    - Hardware interpolation of pixel values
    - Automatic handling of out-of-bounds access
- To work with images you need to create:
  - Image buffers
    - Cfr regular buffers
  - Image samplers
    - To access your image

OpenCL Images
image buffers

Host Code

```
cl_mem clCreateImage(
  cl_context context,
  cl_mem_flags flags,
  const cl_image_format *format,
  const cl_image_desc *image_desc,
  void *host_ptr,
  cl_int *errcode_ret)
```

- Image description:
  - Image dimensions
- Image format:
  - Channel order
  - Channel data type
- OpenCL <= 1.1:
  - clCreateImage1D, clCreateImage2D and clCreateImage3D

Device Code

```
__kernel void manipulateImage(
          __read_only image2d_t src_image,
          __write_only image2d_t dst_image,
          __global sampler_t sampler)
```

- Image:
  - read_only XOR write_only
- Sampler:
  - Necessary to access the image
  - See next

OpenCL Images
image samplers

Host Code

```
cl_sampler clCreateSampler (
    cl_context context,
    cl_bool normalized_coords,
    cl_addressing_mode addressing_mode,
    cl_filter_mode filter_mode,
    cl_int *errcode_ret)
```

- Normalized coordinates:
  - If true: coordinates in [0, 1.0]
- Addressing mode:
  - Behaviour for out of bounds access
- Filter mode:
  - Interpolation behaviour

Device Code

```
__kernel void darkenImage(
           __read_only image2d_t src_image,
           __write_only image2d_t dst_image,
           __global sampler_t sampler)
{
    int2 coord = (int2)(get_global_id(0),
                        get_global_id(1));
    uint offset = get_global_id(1)*0x4000 +
                  get_global_id(0)*0x1000;
    uint4 pixel = read_imageui(src_image,
                               sampler,
                               coord);
    pixel.x -= offset;
    write_imageui(dst_image, coord, pixel);
}
```

OpenCL – OpenGL Interaction
A brief introduction

- Why:
  - Handle computations with OpenCL
    - Typically a lot faster than on the CPU
  - Show result with OpenGL
  - Avoid transfer of data via the host

- What do you need?
  1. An appropriate context
  2. Shared OpenCL and OpenGL data
  3. Synchronization between OpenCL and OpenGL

OpenCL – OpenGL Interaction

Context

- Create a context with the appropriate properties
  - Is platform dependent

```
cl_context_properties properties[] = {
    CL_GL_CONTEXT_KHR, (cl_context_properties) glXGetCurrentContext(),
    CL_GLX_DISPLAY_KHR, (cl_context_properties) glXGetCurrentDisplay(),
    CL_CONTEXT_PLATFORM, (cl_context_properties) platform,
    0};
```

```
cl_context_properties properties[] = {
    CL_GL_CONTEXT_KHR, (cl_context_properties) wglGetCurrentContext(),
    CL_WGL_HDC_KHR, (cl_context_properties) wglGetCurrentDC(),
    CL_CONTEXT_PLATFORM, (cl_context_properties) platform,
    0};
```

```
CGLContextObj glContext = CGLGetCurrentContext();
CGLShareGroupObj shareGroup = CGLGetShareGroup(glContext);

cl_context_properties properties[] = {
    CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE,
    (cl_context_properties)shareGroup,
    0};
```

OpenCL – OpenGL Interaction

Shared data

- OpenGL Vertex Buffer Object ➔ OpenCL buffer

```
cl_mem clCreateFromGLBuffer(cl_context,
                            cl_mem_flags,
                            GLuint, // VBO's unique identifier
                            cl_int *)
```

- OpenGL Texture Object ➔ OpenCL image

```
cl_mem clCreateFromGLTexture(cl_context,
                             cl_mem_flags,
                             GLenum,       // define image type of texture
                             GLint,
                             GLuint,
                             cl_int *)
```

- OpenGL render buffer object ➔ OpenCL image

```
cl_mem clCreateFromGLRenderbuffer(
    cl_context context,
    cl_mem_flags flags,
    GLuint renderbuffer,
    cl_int * errcode_ret)
```

OpenCL – OpenGL Interaction
Synchronization

- Keep OpenCL and OpenGL out of each other’s hairs
- Before running OpenCL kernels:
  - But make sure OpenGL is finished: e.g. glFinish()

```
cl_int clEnqueueAcquireGLObjects (
            cl_command_queue command_queue,
             cl_uint num_objects,
             const cl_mem *mem_objects,
             cl_uint num_events_in_wait_list,
             const cl_event *event_wait_list,
             cl_event *event)
```

- After running OpenCL kernels:
  - But make sure OpenCL is finished: e.g. clFinish()

```
cl_int clEnqueueReleaseGLObjects (
            cl_command_queue command_queue,
             cl_uint num_objects,
             const cl_mem *mem_objects,
             cl_uint num_events_in_wait_list,
             const cl_event *event_wait_list,
             cl_event *event)
```

Advanced OpenCL

- OpenCL is a large topic:
  - Images and OpenGL interoperability
  - Runing code on multiple devices
  - Atomic operations
  - Mapped memory
  - Streaming
  - Events
  - …

- Extend your knowledge as needed.
- But don’t try to run before you can walk!