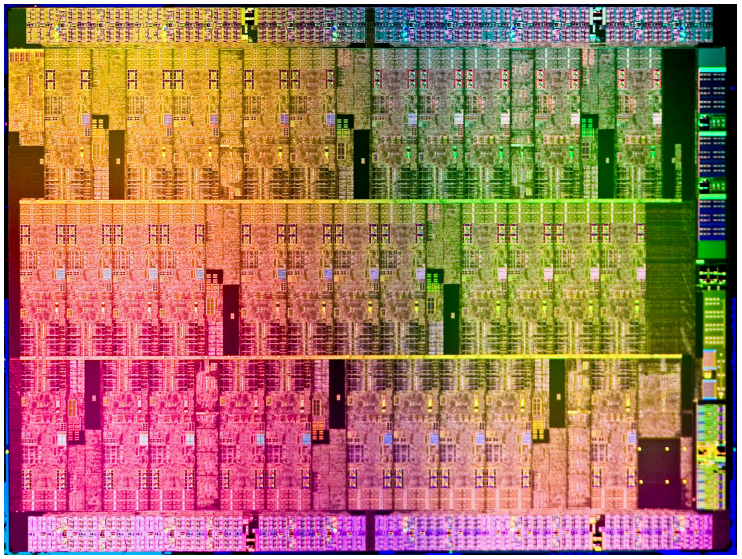
MANCHESTER
1824

The University of Manchester

GPU Computing

Heterogeneous Computing

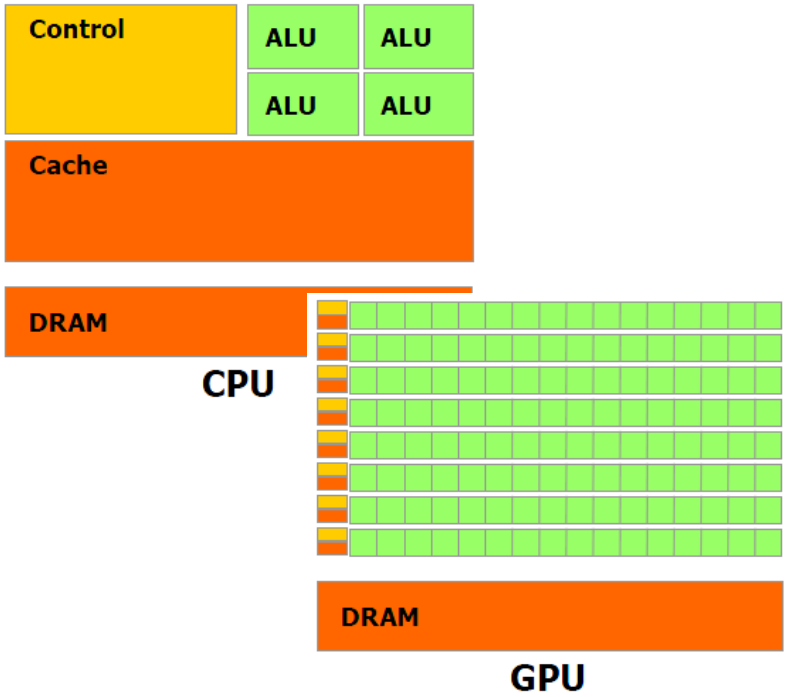
- Modern systems include
  - Multi-core CPUs: ~16 cores
    - *Intel Knights Corner (MIC architecture) +50 cores*
  - GPUs: 100s cores (more later)
    - *CSF: M2050/M2070s 448 core*
  - APUs: CPU+GPU
    - *AMD Fusion*
    - *Nvidia + ARM64 (Project Denver)*
  - Other devices? (IBM Cell, DSP, FPGAs?)
- Why more cores?
  - Split over N cores to get results N times quicker (strong scaling)
  - Split over M cores to for M times larger problems (weak scaling)
  - High Throughput Computing – repeat with different params



Intel MIC architecture (source Intel)

Aims

- Wish to run floating-point intensive programs
  - CPUs: <25% logic devoted to FP.
    - *SSE, AVX? Compiler flags? OpenMP?*
    - *Caches for reducing latency*
  - GPUs: Games, graphics apps? (FP intensive)
  - Other devices (some low-level language?)
- Exploit these heterogeneous resources



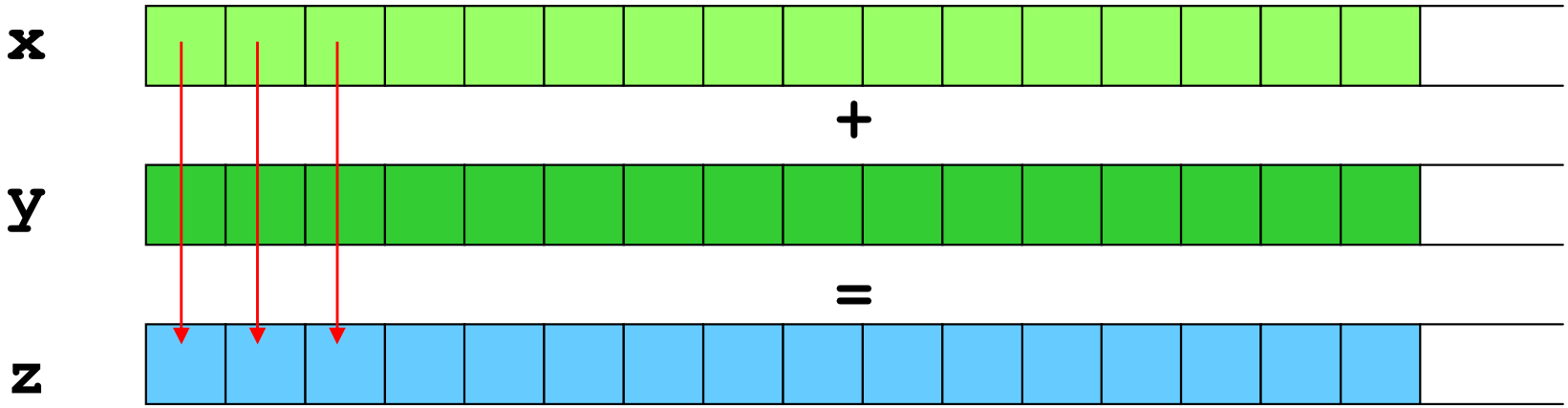
OpenCL – Open Computing Language

Open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. Gives portable and efficient access to heterogeneous resources.

Images by Nvidia, OpenCL Programming Guide, v4.1, 3/1/2012

Simple FP Example: Serial CPU

- Add two vectors of *n* floats: **z = x + y** (large *n*)

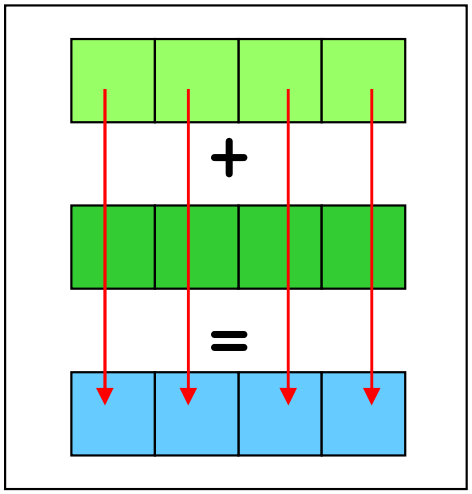



```
void arrayAddCPU( int n, const float *x, const float *y, float *z )
{
  for ( int i=0; i<n; i++ )
    z[i] = x[i] + y[i];
}
```

- Exploit the fact that iteration i independent of i+1

Vector Processing in CPU

- Extensions added to CPUs: SSE*n*, AVX
  - H/W vector units: do more FP ops in parallel
- Must have *data parallelism* / independence
- Single Instruction Multiple Data (SIMD)
  - OPs performed in *lock-step*



128bit wide: 4 x 32bit Float vector (SIMD) unit

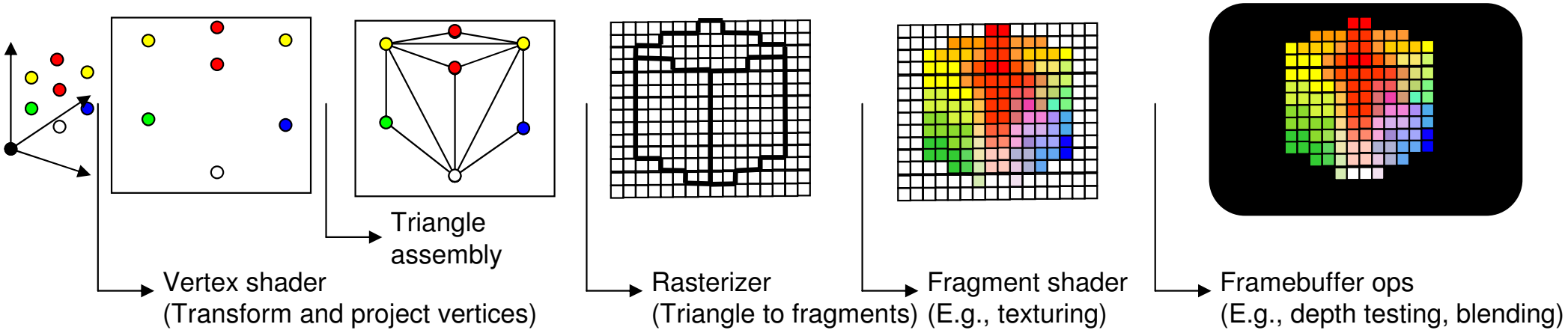
```
#include <xmmintrin.h>
void arrayAddSSE( int n, const float *x, const float *y, float *z )
{
  __m128 sse_x, sse_y, sse_z;
  for (int i=0; i<n/4; i++ ) {
    sse_x = _mm_load_ps(x);
    sse_y = _mm_load_ps(y);
    sse_z = _mm_add_ps(sse_x, sse_y);
    _mm_store_ps(z, sse_z);
    x+=4; y+=4; z+=4;
  }
}
```

Based on SSE code by Jacques du Troit, NAG.

Why GPUs?

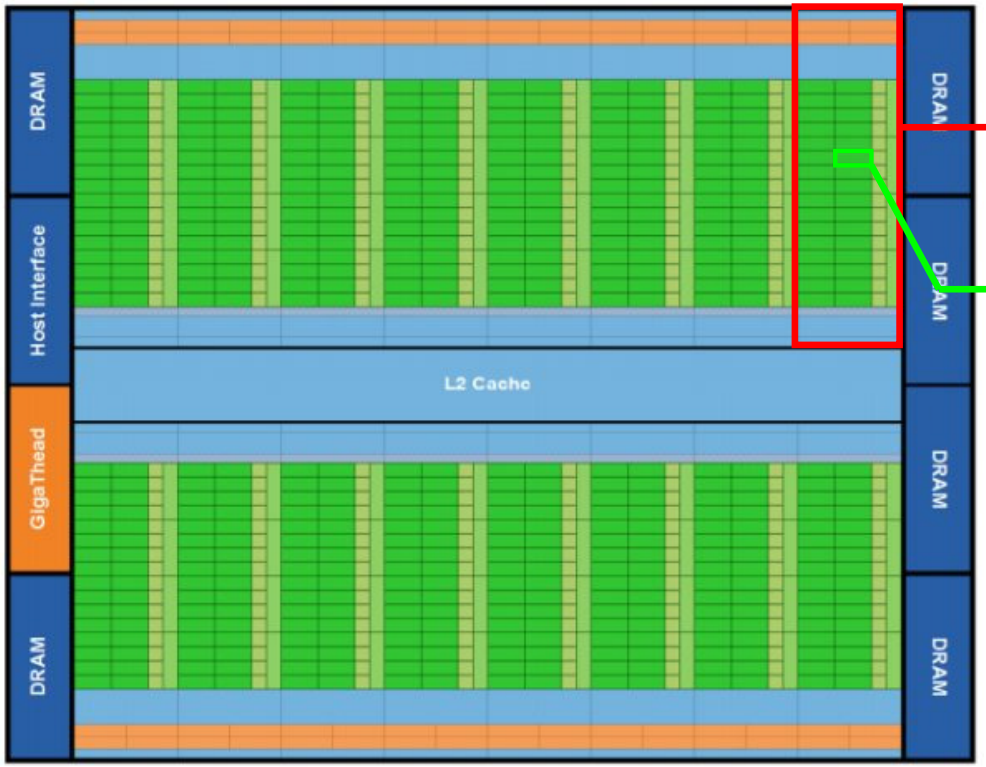
Why all the talk of GPUs?

- Graphics rendering: highly data parallel
- Transform geometry: E.g., 4x4 matrix mult with *n* vertices
  - *A vertex is [x, y, z, 1.0] – a vector of floats*
- Manipulate fragments / pixels: E.g., 1280x1024 pixels
  - *A pixel is [r, g, b, a] – a vector of floats*
- GPU h/w: vector processors



GPU Architecture

- View GPU as vector processor or a multi-core processor?
  - E.g.: Nvidia Fermi architecture (GeForce, Quadro and Tesla)



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

- 512 cores arranges as:
  - 16 x Streaming Multiprocessors
    - *An SM is a SIMD vector processor*
  - 32 cores per SM
  - 16 x 32 = 512 cores
  - Each SM has
    - *scheduler / dispatch*
    - *Register file, cache: L1+shm (64KB), L2 (768KB)*

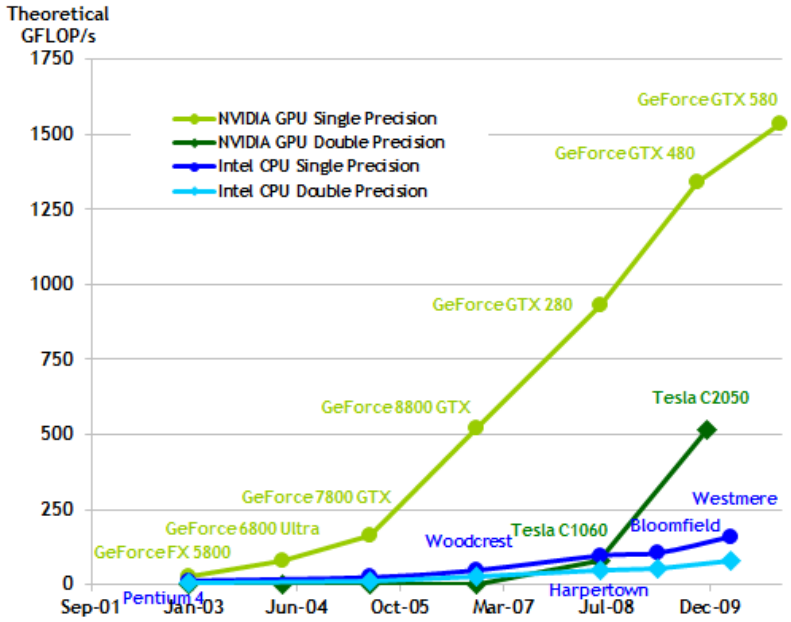
Code runs on cores

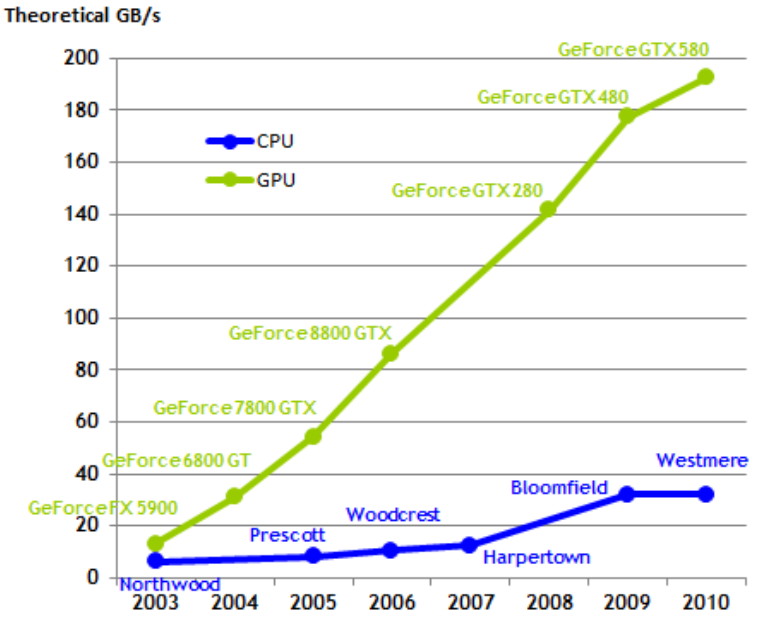
- For now think of having *n* cores
- Keep in mind the vector units

Image by Nvidia, Fermi White Paper v1.1

GPUs

- Summary: GPUs suited to highly data-parallel problems
  - Many cores arranged as vector processing h/w
  - Programmable
  - Fast memory (>150GB/s)

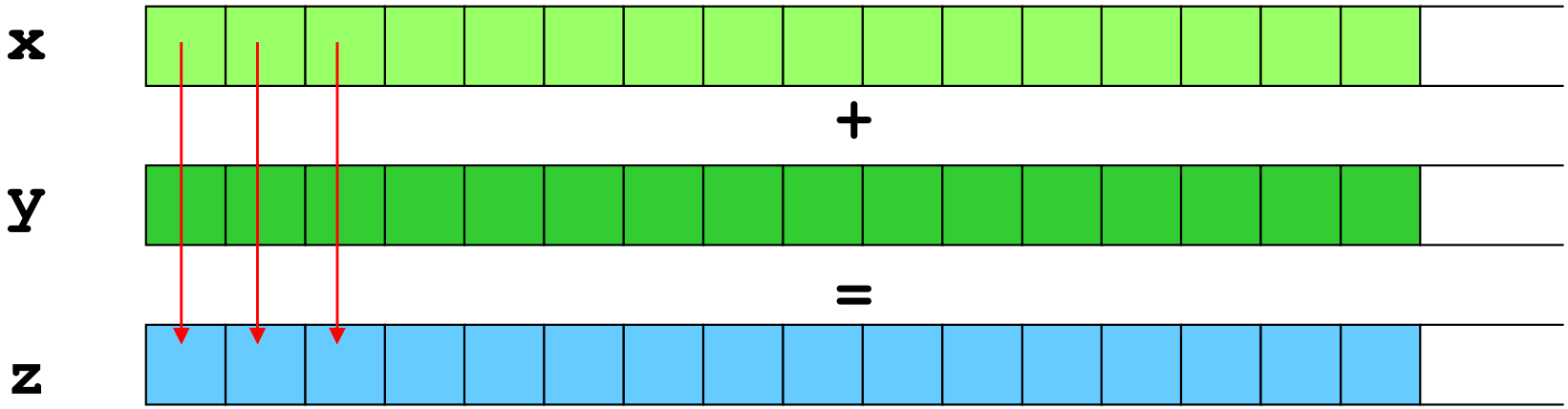




Graphs by Nvidia, OpenCL Programming Guide, v4.1, 3/1/2012

A bit of OpenCL

- The OpenCL way: replace loops with a kernel
  - A *kernel* is run on every data item in parallel on the device (GPU)
    - *Similarities with the previous vector and shader examples!*




```
void
arrayAddCPU( const int n,
             const float *x,
             const float *y,
             float *z )
{
  int i;
  for ( i=0; i<n; i++ )
    z[i] = x[i] + y[i];
}
```

```
__kernel void
arrayAddOCL(
             __global const float *x,
             __global const float *y,
             __global float *z )
{
  int i = get_global_id(0);

  z[i] = x[i] + y[i];
}
```

MANCHESTER 1824

The University of Manchester

OpenCL Basics

Background

- Created by Khronos Compute Group + industry ~2008
  - Apple wanted many-core CPU, GPU standard
  - Specification to be implemented by vendors
    - *OpenCL 1.0 (Dec 2008), 1.1 (Jun 2010), 1.2 (Nov 2011)*
  - Currently most vendors support up to v1.1
- Resources: OpenCL Registry: *http://www.khronos.org/opencl/*
  - Specification (READ THIS)
    - *www.khronos.org/registry/cl/specs/opencl-1.1.pdf*
    - *www.khronos.org/registry/cl/specs/opencl-1.2.pdf*
  - Reference card (REFER TO THIS)
    - *http://www.khronos.org/files/opencl-1-1-quick-reference-card.pdf*
    - *http://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf*
  - Reference (man) pages:
    - *http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/*
    - *http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/*

Books

- The OpenCL Programming Book

  Free HTML version at: http://www.fixstars.com/en/opencl/book

- OpenCL in Action: How to accelerate graphics and computation

  Scarpino. Manning Publications (Nov '11)

- Heterogeneous Computing with OpenCL

  Gaster, Howes, Kaeli, Mistry, Schaa. Morgan Kaufmann (Oct '11)

- OpenCL Programming Guide

  Munshi, Gaster, Mattson, Fung, Ginsburg. Addison Wesley (July '11)

Anatomy of OpenCL

- Language Specification (use to write kernels)
  - C-based subset of ISO C99 (+ language extensions)
  - IEEE 754 numerical accuracy
  - Online / offline compilation of compute kernels
- Platform Layer API (use from the *host*)
  - Hardware abstraction layer over your various devices
  - Query, select, init compute devices
  - Create compute contexts and work-queues
- Runtime API (used to queue up and run your kernels)
  - Execute compute kernels
  - Manage scheduling, compute and memory resources

OpenCL Platform Model

- The OpenCL view of your system

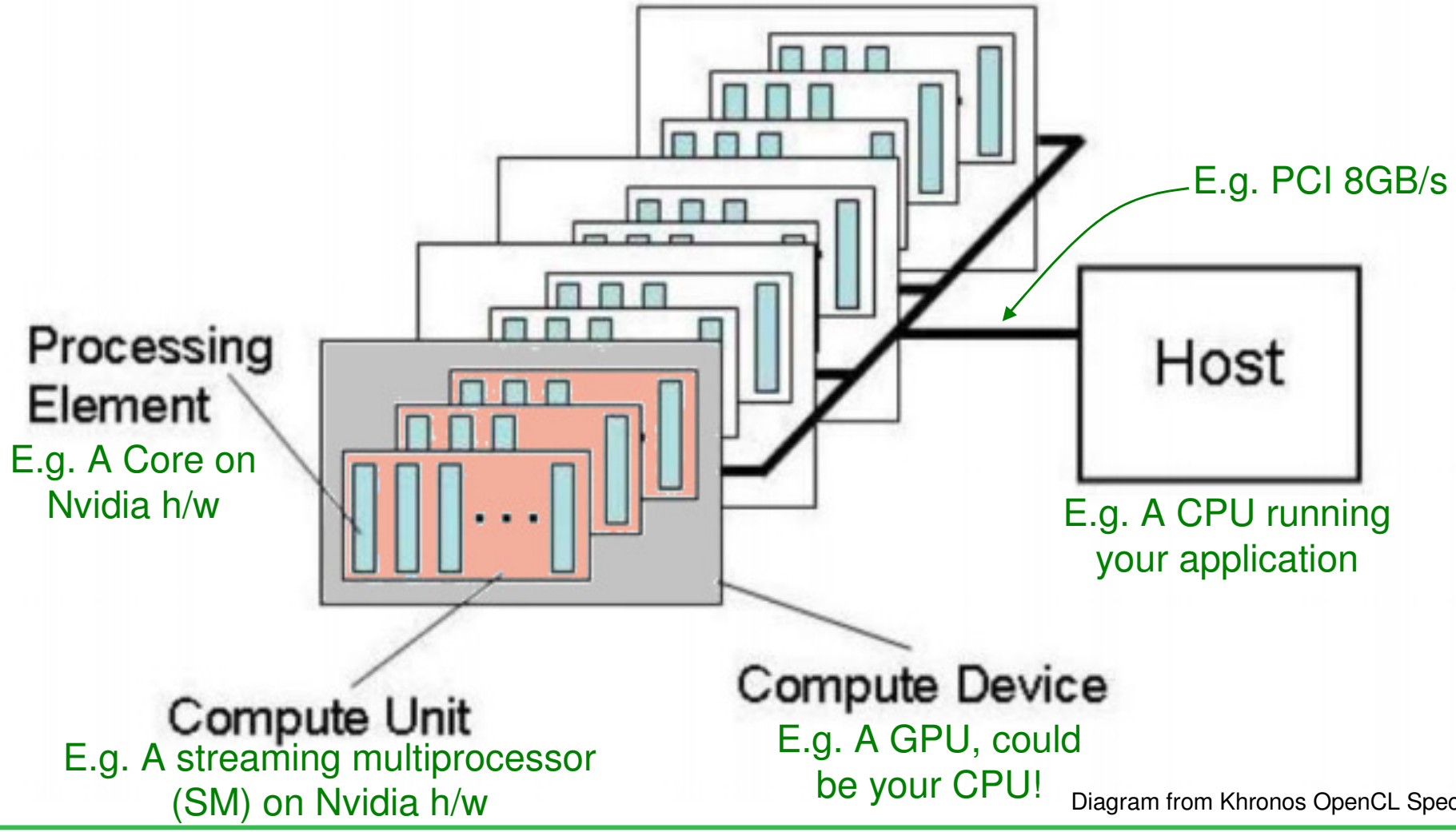


Diagram from Khronos OpenCL Specification

OpenCL Execution Model

- Expression of parallelism for data-parallel execution
- A *kernel* (function) executes on an OpenCL device.
  - Written in OpenCL C
  - Many instances of a kernel execute in parallel but on different data
- A *work-item* is an instance of the kernel
  - Executed by a Processing Element (PE) in the device
  - Given a unique id to distinguish it from other work-items
- A *work-group* is a collection of work-items
  - Grouped to support your algorithm, sharing local memory, barriers
  - GPU requires batches of work-items (num WI's >> num cores)
  - Its work-items execute concurrently on the same Compute Unit

Expressing Parallelism

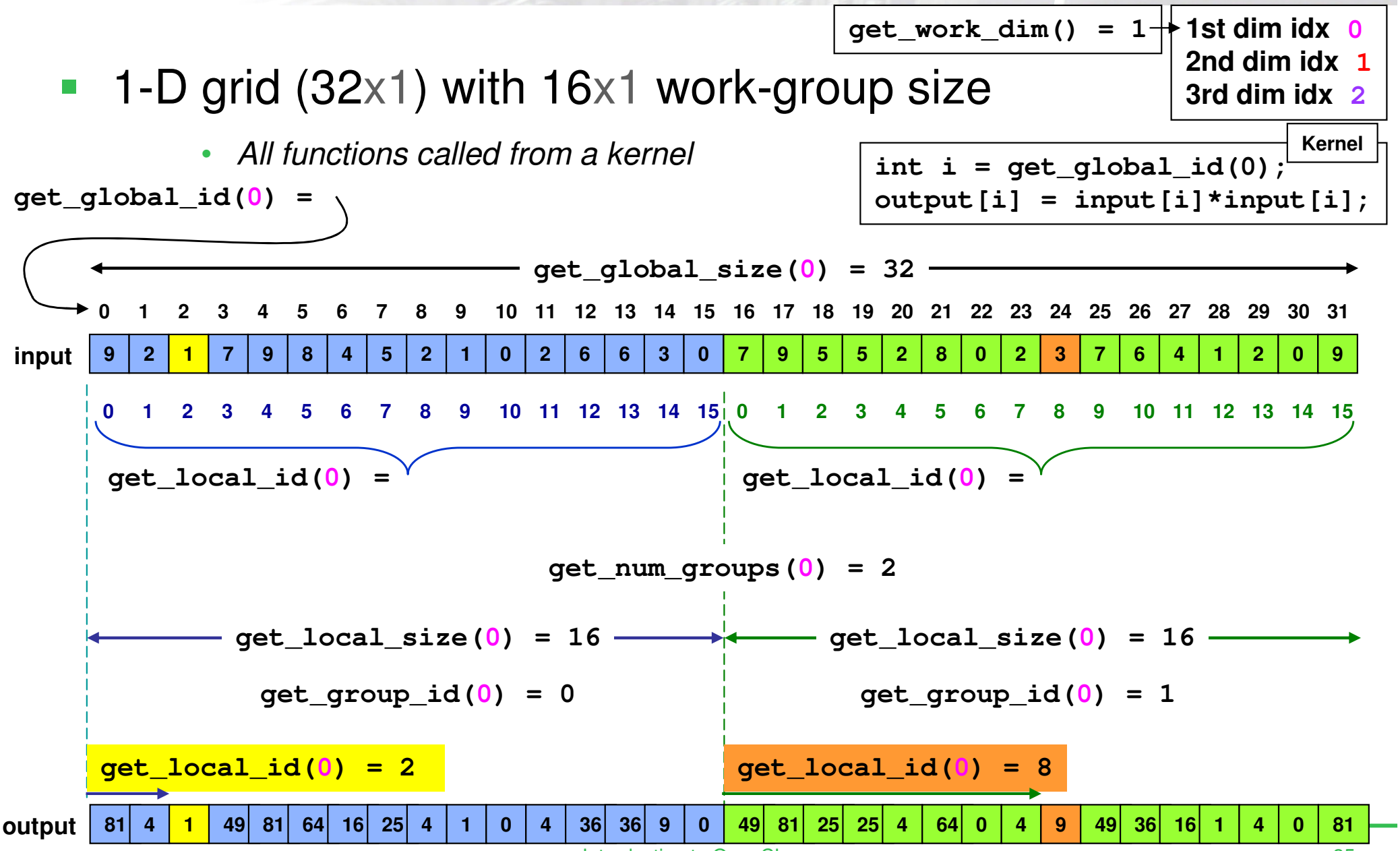
- Define our problem domain as an *N*-Dimensional domain
  - N = 1, 2 or 3. Examples:
    - *1D: Add two vectors (already seen)*
    - *2D: Process an image (e.g., blur) 2k x 2k pixels = 4M kernel execs*
    - *3D: Process volumetric data acquired from scanner*
- Execute kernel at each point in the problem domain
  - Sometime called a *work-grid* or *index space*
  - No creating threads within kernel – write your code for a single thread (more like MPI than OpenMP?)
- Choose dimension to match your algorithm, thought process …
  - can do a 2D or 3D problem using 1D indexing

Work-items and Work-groups

- Each instance of a kernel execution is a *work-item*
  - CUDA calls this a thread and refers to SIMT
  - Each work-item given a unique global id
  - Same code executed for every work-item
    - *Path through code can vary (if / else etc)*
- Work-items (threads) are grouped in to work-groups
  - Allows scheduling of device resources, providing scalability
  - Each work-group given a unique group id
  - Each work-item within a work-group given a unique local id
- Built-in functions for kernel to get the various ids
  - A work-item's kernel instance can use these to figure out which data elements to work with…but it is *your* code that does this!

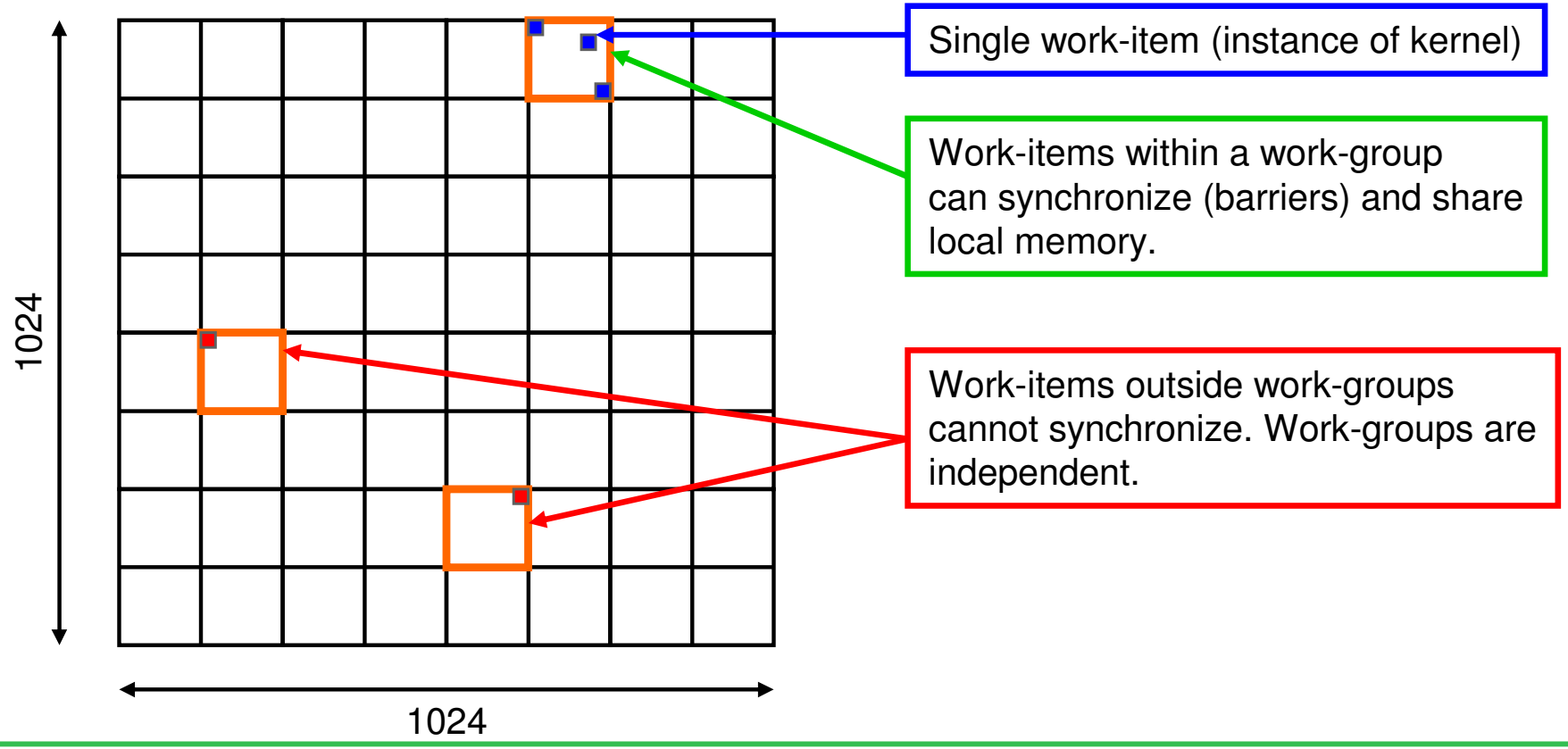
1-D Example

- 1-D grid (32x1) with 16x1 work-group size
  - *All functions called from a kernel*



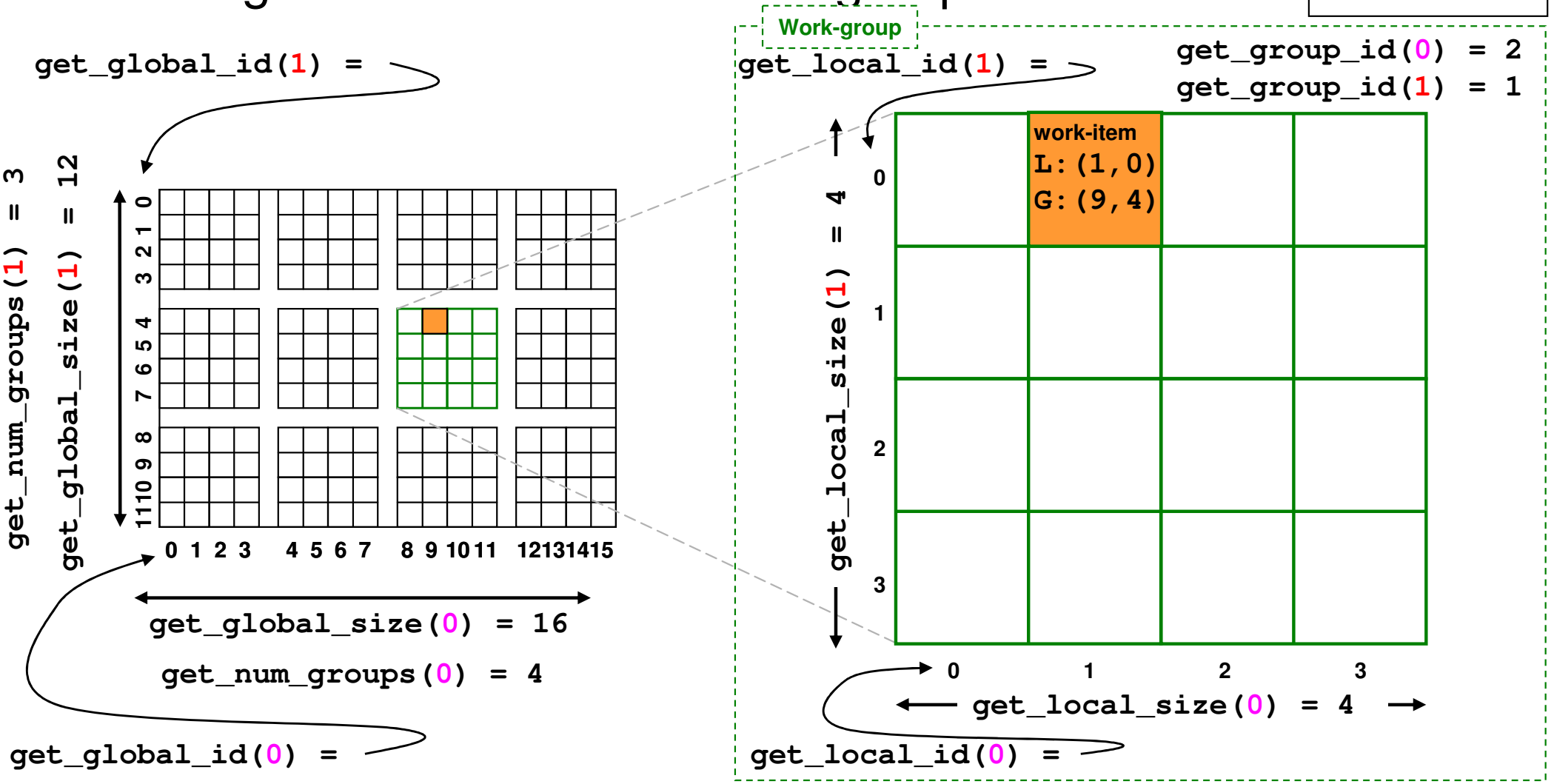
2-D Example

- Example 2-D grid of work-items (e.g., image processing)
  - Global dimensions: 1024 x 1024 (the entire problem domain)
  - Local dimensions: 128 x 128 (work-group size)



2-D Example

- 2-D grid 16x12 with 4x4 work-group size



Work-groups

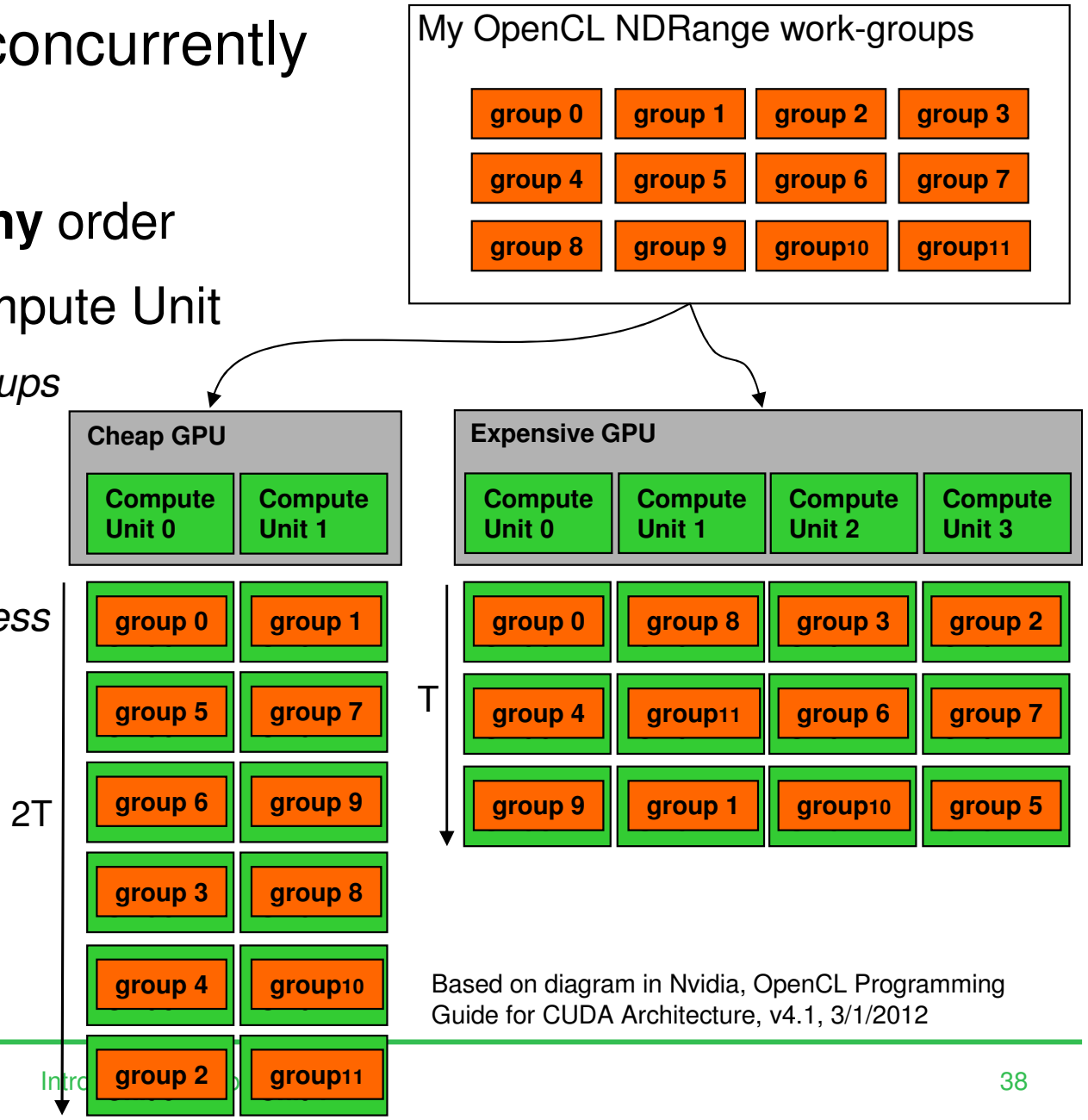
- Work-items (threads) within a work-group execute concurrently, at least conceptually
  - All work-groups have same number of work-items
    - *H/w specifies maximum number allowed*
    - *Cannot change number of work-items once kernel is running*
- You specify number of work-items (global dimensions)
  - And *optionally* number of work-items per work-group (i.e., local dimensions)

```
// Work sizes to process our data (e.g., 2k rows x 4k cols array)
int nDim = 2;
int globalSize[2] = {4096, 2048}; // {x,y} not {rows,cols}
int localSize[2]  = {32, 32};     // 1024 threads in our work-groups
// Launch kernel specifying nDim, globalSize, localSize (see later)
```

  - OpenCL calculates number of work-groups
  - The local-size must divide evenly the global size in each dimension
    - *Different to CUDA which requires num work-groups* and *work-group size*

Work-groups = Scalable Programming Model

- *n* work-groups execute concurrently
  - Depends on device h/w
  - Work-groups are run in **any** order
  - Work-group sent to a Compute Unit
    - *More CUs = more work-groups running in parallel*
    - *All work-groups eventually processed*
    - *E.g. 4M work-items to process a large array executed by a ~500 core GPU*
  - Provides scalability
    - *Automatic speed-up*
    - *Linear, in theory*



Based on diagram in Nvidia, OpenCL Programming Guide for CUDA Architecture, v4.1, 3/1/2012

Scalable Programming

- You'll usually have *num* work-items >> *num* GPU cores
  - and *num* work-groups >> *num* Compute Units
- GPU swaps work-groups (and their work-items) between being active and inactive
  - Usually when work-items start waiting for data from memory
  - Get other work-items working while some are waiting
- You are hiding the delay (latency) in memory access with more work
- Lots of work-items is good!

MANCHESTER 1824

The University of Manchester

Writing an OpenCL Program

Host Code

Our code

- What code are we going to develop?
- Host program
  - Set up your device (GPU etc)
  - Transfer data to/from the device
  - "Manage" kernels to be run on the device (see later)
  - Don't try to turn all your code in to a kernel!
- Device (GPUs) run your Kernel(s)
  - Execute on each "data point" in the problem domain in parallel
  - Performs your computation
    - *Usually a small function. Several different kernels used for large problems.*
  - Read / write device memory
    - *Think about which memory space a variable exists in (see later)*

Our OpenCL App

- OpenCL skeleton (host code)

```
// 0. CPU application setup code (read files etc)

// OpenCL Setup
// 1. Set up OpenCL platform (AMD, Nvidia, Intel's OpenCL)
// 2. Set up OpenCL device(s) supported by platform (GPU,CPU,...)
// 3. Create OpenCL context to manage device(s), memory objects...
// 4. Create OpenCL command queues for device(s) in context
// 5. Create OpenCL program from kernel sources in context
// 6. Use OpenCL to build/compile source for the device(s) in context
// 7. Extract required kernel from compiled program (may be >1)

// Parts of your application that has been OpenCL-ised
// 8. Allocate memory objects on the device
// 9. Transfer data from host to device memory objects
// 10. Specify kernel arguments (input args)
// 11. Launch extracted kernel via device's command queue
// 12. Copy data back from device via device's command queue
// Repeat 8-12 usually

// Other optional CPU work

// 13. Tidy up OpenCL objects
```

Summary of the steps

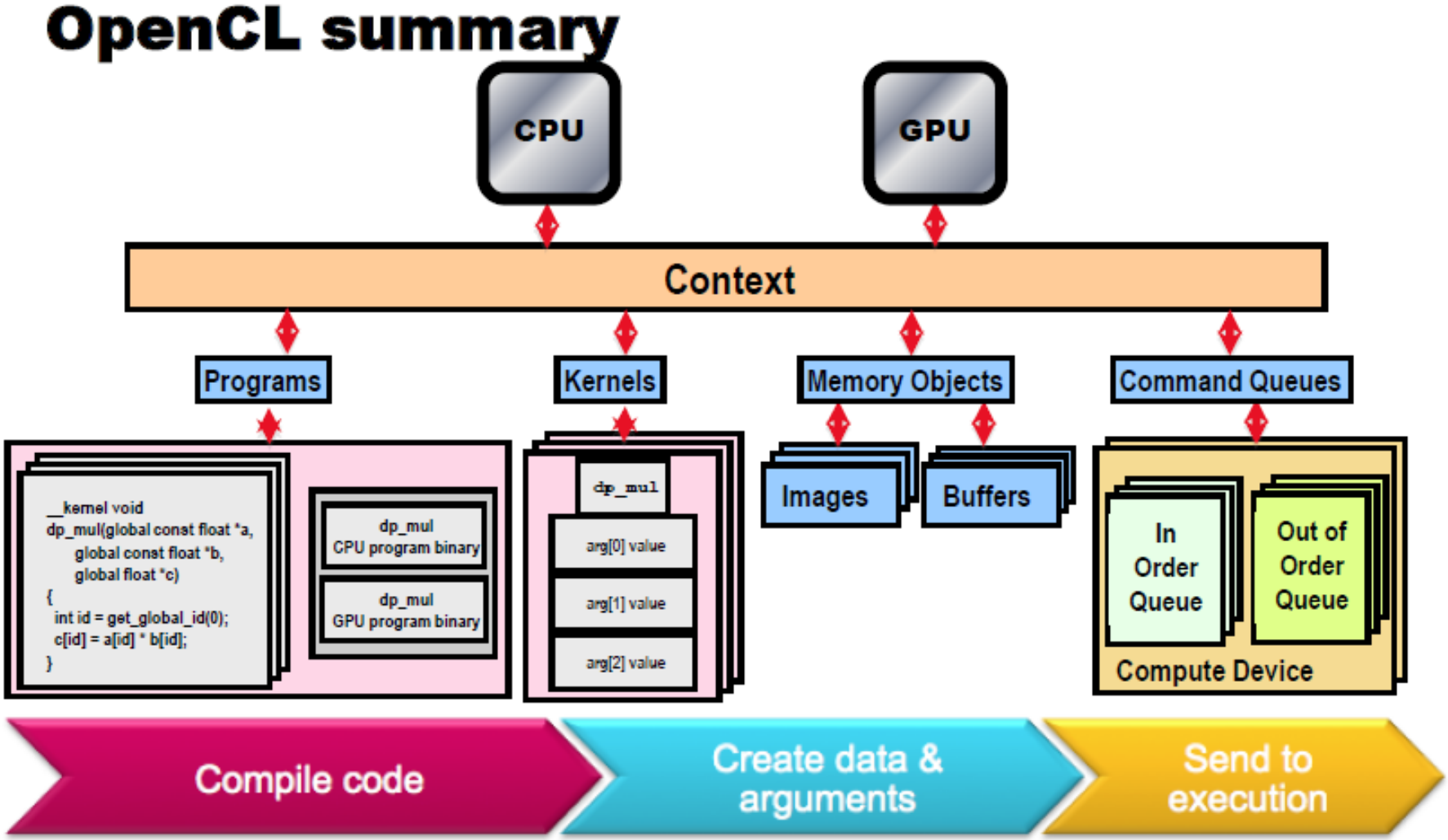


Diagram by Tim Mattson (Intel), Udeepta Bordoloi (AMD), OpenCL An Introduction for HPC programmers, ISC11 presentation

1.Setting up on the Host – Get a Platform

- Select a Platform (often just the one in your system)
  - Nvidia, AMD, Intel etc. Like selecting the correct libOpenCL.so/.dll

```
cl_int clGetPlatformIDs( cl_uint num_entries,      cl_platform_id *platforms,
                         cl_uint *num_platforms )
```

```
#include <CL/opencl.h>
int main( void )
{
  cl_int          err;              // OpenCL funcs give an error flag
  cl_uint         num_platforms;
  cl_platform_id platform_id;

  // Assuming just one platform (not portable)
  err = clGetPlatformIDs( 1, &platform_id, &num_platforms );

  if ( err != CL_SUCCESS || num_platforms == 0 ) {
    // No platforms. Exit!
    printf( "No OpenCL platforms!\n" );
    return 1;
  }
  ...
}
```

OpenCL API Errors

- Platform API functions return an error code
  - Function return value or output arg
  - cl_int, several error values available
    - *Errors if you supply bad function args*
    - *Errors if out of resources on host or device*
  - At least check for CL_SUCCESS then narrow it down
- Example: `err = clGetPlatformIDs(0, &platform_id, &num_platforms);`

CL_SUCCESS	Function executed successfully
CL_INVALID_VALUE	If *num_entries* is zero and *platforms* is not NULL or if both *num_platforms* and *platforms* are NULL
CL_OUT_OF_HOST_MEMORY	If there is a failure to allocated resources required by the OpenCL implementation on the host.

More than one Platform?

- A rare case? Might get NV/AMD + Intel on CSF (in future)
  - Ask OpenCL for number of platforms, get them, query them

```
cl_int clGetPlatformInfo( cl_platform_id platform,  cl_platform_info param_name,
                          size_t param_value_size,  void *param_value,
                          size_t *param_value_size_ret )
```

```
int             i;
cl_uint         num_platforms;
cl_platform_id *platform_ids;    // Will be an array of ids
char            pinfo[256];      // We're going to ask for a string property

// Get number of ids then the actual ids in to an array
err = clGetPlatformIDs( 0, NULL, &num_platforms );
platform_ids = (cl_platform_id *)malloc(num_platforms*sizeof(cl_platform_id));
err = clGetPlatformIDs( num_platforms, platform_ids, NULL );

// Get info about each platform
for ( i=0; i<num_platforms; i++ ) {
  err = clGetPlatformInfo( platform_ids[i], CL_PLATFORM_VENDOR, sizeof(pinfo),
                           pinfo, NULL ); // Ignoring the return size
  if (!strcmp(pinfo, "Advanced Micro Devices, Inc."))
    break;      // Stop searching. Use platform_ids[i] from here on.
}
```

2. Get a device

- Require the devices supported by the platform
  - GPUs, CPU, generic accelerator – depends on vendor

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

```
CL_DEVICE_TYPE_GPU
CL_DEVICE_TYPE_CPU
CL_DEVICE_TYPE_ACCELERATOR
CL_DEVICE_TYPE_DEFAULT
CL_DEVICE_TYPE_ALL
```

```
...

cl_device_id    device_id;

// Assuming we just want the first GPU device
err = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL );
if ( err != CL_SUCCESS ) ...

// Alternatively, get all devices (then choose one or more)
cl_uint         num_devices;
cl_device_id    *device_ids;      // Will be an array of ids

clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 0, NULL, &num_devices );
device_ids = (cl_device_id *)malloc( num_devices*sizeof(cl_device_id) );
clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, num_devices, device_ids,NULL);
```

Choose a Device

- Can query a device for h/w details
  - See the OpenCL spec for param names

```
cl_int clGetDeviceInfo(   cl_device_id device,
                          cl_device_info param_name,
                          size_t param_value_size,
                          void *param_value,
                          size_t *param_value_size_ret )
```

```
int             j;
cl_uint         dinfo;              // We're going to ask for a numeric property

for ( j=0; j<num_devices; j++ ) {
  err = clGetDeviceInfo( device_ids[j], CL_DEVICE_MAX_COMPUTE_UNITS,
                         sizeof(dinfo), &dinfo, NULL ); // Ignoring return size

  // Choose device based on properties (somehow - up to you)
  if ( dinfo > MY_MINIMUM_PROPERTY_VALUE )
    break;      // Stop searching. Use this device_ids[j] from here on.
}
```

The Context

- The Context is an environment / container on the host
- Manages OpenCL objects and resources. It contains
  - Device(s) from a single platform (AMD or Nvidia or …)
  - Memory objects on the devices
    - *Devices in same context can share memory objects*
  - Programs (collection of kernels and functions to run on the device)
  - Command queues: host interacts with a device via a queue
    - *Memory commands (transfers to/from the device)*
    - *Kernel launches*
    - *Events / synchronisation of commands*

3. Create a context

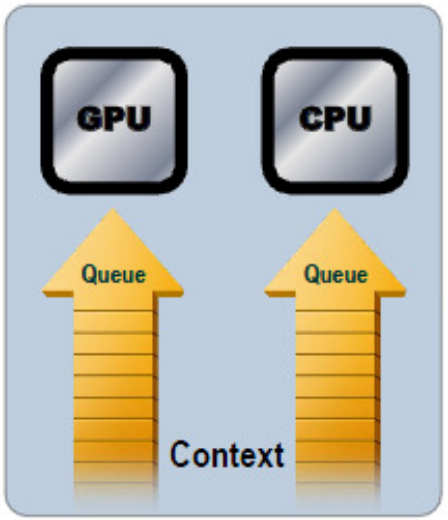
- Context contains one or more devices
  - Nasty looking function but we ignore most of the args

```
cl_context clCreateContext( const cl_contextproperties *properties,
                            cl_uint num_devices,
                            const cl_device_id *devices
                            void (CL_CALLBACK *pfn_notify)(...),
                            void *user_data,
                            cl_int *errcode_ret )
```

```
...
cl_context              ctx;              // Stores our new context
cl_context_properties   ctx_props[3];     // Store required properties in array

ctx_props[0] = CL_CONTEXT_PLATFORM;       // Name of property
ctx_props[1] = platform_id;               // Value of property (from earlier)
ctx_props[2] = 0;                         // Must end list with 0
                                          // (Odd way of supplying a platform_id)
// Just the one device in our context
ctx = clCreateContext( ctx_props, 1, &device_id, NULL, NULL, &err );

// Alternatively, multiple devices (maybe GPUs + CPU etc) in our context
ctx = clCreateContext( ctx_props, num_devices, device_ids, NULL, NULL, &err );
```

Command Queues

- Each device requires its own command queue
  - Queue points to one device (in a context)
    - *Typically only one queue and one device associated*
  - A device can be attached to multiple queues
    - *Allows processing of independent async commands*
  - Host puts commands (operations) in to the queue
    - *This is how you ask the GPU to do something*
    - *All commands go via the queue. Like tasks.*
  - Device executes the commands
    - *in-order : order in which you put commands in queue (default)*
    - *out-of-order : device schedules commands as it sees fit*
  - Can synchronize within and across queues
    - *e.g., prevent one command running before another completes*



4. Create a Command Queue

- Can't talk to our device without one
  - Two properties: CL_QUEUE_OUT_OF_ORDER_EXEC_MODE
    CL_QUEUE_PROFILING_ENABLE

```
cl_command_queue clCreateCommandQueue( cl_context context,
                                       cl_device_id device,
                                       cl_command_queue_properties properties,
                                       cl_int *errcode_ret )
```

```
...
cl_command_queue                cmd_queue;      // Stores our new command queue
cl_command_queue_properties     q_props;        // Bitfield: OR any flags

q_props = 0;    // Will use queue defaults (in-order, no profiling)

// Repeat this function for each device in our context (ctx)
cmd_queue = clCreateCommandQueue( ctx, device_id, q_props, &err );

// E.g., if we have multiple devices (from earlier) and want a queue per device
cl_command_queue *cmd_queues;
cmd_queues = (cl_command_queue *)malloc(num_devices*sizeof(cl_command_queue));
for ( q=0; q<num_devices; q++ ) {
  cmd_queues[q] = clCreateCommandQueue( ctx, device_ids[q], 0, &err );
}
```

OpenCL Programs

- OpenCL Program is a set of kernels (and functions)
  - Kernels, functions written in OpenCL C source
  - Compiled at runtime by the OpenCL driver
    - *i.e., while your application is running, OpenCL will compile your kernel code*
  - Require the kernel source code at runtime
    - *As a string (or array of strings)*
    - *You could create this by reading from a text file*
  - Could also supply a compiled binary
    - *See clCreateProgramWithBinary() and clGetProgramInfo(..., CL_PROGRAM_BINARIES, ...) (not covered today)*
  - This is all a bit messy
    - *However, kernels are compiled **for your specific device***
    - *You only need a C compiler, for host code (no additional compiler ala CUDA)*

5. Create a Program object

- Supplying OpenCL C source in host code (nasty)

```
cl_program clCreateProgramWithSource( cl_context context,
                                      cl_uint count,
                                      const char **strings,
                                      const size_t *lengths,
                                      cl_int *errcode_ret )
```

```
// An array of 5 strings in your host code (very nasty - see next slide)
const char *ksource1[] = {
  "  __kernel void arraySqr(__global float *a, unsigned int n ) {  ",
  "    unsigned int idx = get_global_id(0);                        ",
  "    if ( idx < n )                                              ",
  "      a[idx] = a[idx]*a[idx];                                   ",
  "  }                                                             ",
};

cl_program     prog;              // Holds ours new program object

// No string lengths supplied. Ensure strings are NULL terminated (true above)
prog = clCreateProgramWithSource( ctx, 5, (const char **)ksource1, NULL, &err);
```

Read Source from File

- Write kernels etc in text file (often named `.cl`)

```
cl_program      prog;                                   // Holds our new program

const char      *fname = "mykernel.cl";                 // Vars to read a file
FILE            *kfile;
size_t          kfilesize;
char            *ksource;

// Read a text file in to a string
kfile = fopen( fname, "r" );                            // Open .cl file for reading
fseek( kfile, 0, SEEK_END );                            // Go to end of file
kfilesize = (size_t)ftell(kfile);                       // Get position (i.e., length)
rewind( kfile );                                        // Go back to start
ksource = (char *)malloc(kfilesize*sizeof(char));       // Alloc one long string
fread(ksource, 1, kfilesize, kfile );                   // Read file in to string
fclose( kfile );                                        // Finished with source file

// One string containing all the source code read in, length passed in.
prog = clCreateProgramWithSource( ctx, 1, (const char **)&ksource, &kfilesize,
                                  &err);
```

6. Building your Kernel source

- Ask OpenCL to build (compile & link) the source
  - Compiled for specific devices. Looks nasty (but ignore most args)
  - Supply compiler flags in a string

```
cl_int clBuildProgram(    cl_program program,
                          cl_uint num_devices,
                          const cl_device_id *devices,
                          const char *options,
                          void (CL_CALLBACK *pfn_notify)(...),
                          void *user_data )
```

```
// Build source for all devices associated with the program
// or supply a list of specific (one or more) devices
err = clBuildProgram( program, 0, NULL, NULL, NULL, NULL );
if ( err != CL_SUCCESS ) ... // CHECK THE ERROR - INDICATES COMPILATION FAILED

// Usual compiler options (-I for include dirs, -D for #ifdef, #define macros)
// There are vendor-specific options (e.g., Nvidia: -cl-nv-verbose)
char options[] = "-I/my/include/dir -DMY_FLAG=something";
err = clBuildProgram( program, 0, NULL, options, NULL, NULL );
// Or supply a simple const string
err = clBuildProgram( program, 0, NULL, "-DMY_SIZE=1000", NULL, NULL );
```

When it doesn't Compile

- Very easy to make mistakes in kernel source
  - Not easy to get compiler output and helpful messages
  - Might need several attempts at compiling simpler code

```
cl_int clGetProgramBuildInfo( cl_program program,
                              cl_device_id device,
                              cl_program_build_info param_name,
                              size_t param_value_size,
                              void *param_value,
                              size_t *param_value_size_ret )
```

```
size_t          log_size;       // Length of build log
char            *build_log;     // String containing the build log

// Use CL_PROGRAM_BUILD_LOG for the param name. Get the log size first.
err = clGetProgramBuildInfo( program, device_id, CL_PROGRAM_BUILD_LOG,
                             0, NULL, &log_size );
build_log = (char *)malloc(log_size*sizeof(char));
err = clGetProgramBuildInfo( program, device_id, CL_PROGRAM_BUILD_LOG,
                             log_size, build_log, NULL );
printf( "Build log\n%s\n", build_log );
```

7. Extract required Kernel for Launching

- Program now contains one or more compiled kernels
  - Get a kernel object from the program so we can run it
  - Use same kernel name from the source code

```
cl_kernel clCreateKernel( cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret  )
```

```
cl_kernel       kernel;                 // Holds our new kernel
const char      *kname = "arrayAdd";    // Name of the kernel

// Get the named kernel
kernel = clCreateKernel( program, kname, &err );

// May have compiled several kernels
cl_kernel       kernel1, kernel2;
kernel1 = clCreateKernel( program, "arrayAdd", &err );
kernel2 = clCreateKernel( program, "arraySum", &err );
```

Steps 1-7 Done

- Summary of Setup
  - Got a platform (Nvidia, AMD, etc)
  - Got device(s) supported by that platform (GPUs …)
  - Created a context to manage devices and other objects
  - Created a command queue, in that context, to feed device
  - Created a program, in that context, from OpenCL C source code
  - Built (compiled and linked) the program for device(s) in the context
  - Extracted a specific kernel so we can launch it on the device

Steps 8…13

- Now some more interesting work
  - Allocate memory objects on the device
  - Transfer data from host to device
  - Specify the arguments for a kernel launch
  - Launch the kernel
  - Read data back from the device (results of our kernel)
- Repeat as required
  - May have other kernels to launch
- Cleanup objects at end

Memory Objects / Buffers

- Data from host must be copied to the device
  - GPU device has physical DRAM distinct from host
  - Buffer objects used as containers
    - *Managed by the context and visible to the devices in that context*
    - *If using CPU devices, still require buffer objects*
- Create the buffer then populate it
  - Can do this in one step or separately
- Transferring data to devices is costly
  - Transfer over PCIe bus (e.g., current GPUs)
  - This can kill any performance gain
- Kernels take mem objs as inputs and outputs

8. Create a Memory Object

- Specify type of access and size
  - Buffers are 1D hence calc your size correctly (number of bytes)
  - Can contain scalars (int, float,…), vectors (float4, …), user structs

```
cl_mem clCreateBuffer( cl_context context, cl_mem_flags flags,
                       size_t size, void *host_ptr,
                       cl_int *errcode_ret  )
```

```
long           count;          // Example number of elements in my array
float          *vecA_h;        // Often use a _h suffix for host mem
cl_mem         vecA_d;         // Often use a _d suffix for device mem

// Create a write-only buffer: kernel will only write it, not read it
vecA_d = clCreateBuffer( ctx, CL_MEM_WRITE_ONLY, sizeof(float)*count,
                         NULL, &err );

// Flags OR-ed together: kernel will only read buffer. We populate buffer too.
cl_mem         vecB_d;
float          *vecB_h = readFloatDataFromFile("myinput.dat", &count);
cl_mem_flags   mflags = CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR;
vecB_d = clCreateBuffer( ctx, mflags, sizeof(float)*count, vecB_h, &err );
```

Buffer flags

- cl_mem_flags bitfield

CL_MEM_READ_WRITE	Allocate a mem obj that can be read from **and** written to by a kernel.
CL_MEM_WRITE_ONLY	Mem obj **not** read by kernel. Reading is undefined. Cannot use with CL_MEM_READ_WRITE.
CL_MEM_READ_ONLY	Mem obj is read-only when used in a kernel. Writing to within kernel is undefined. Cannot use with other WRITE flags.
CL_MEM_USE_HOST_PTR	Use storage bits referenced by *host_ptr* arg (cannot be NULL) as mem obj. OpenCL may cache content in device memory for use by kernels.
CL_MEM_ALLOC_HOST_PTR	Asks OpenCL to alloc memory for mem obj from host memory. Cannot use with CL_MEM_USE_HOST_PTR.
CL_MEM_COPY_HOST_PTR	Ask OpenCL to alloc device memory and copy data in from *host_ptr*. Cannot be used with CL_MEM_USE_HOST_PTR. **Can** be used with CL_MEM_ALLOC_HOST_PTR.

9. Copying data via Command Queue

- Can read contiguous data from device to host or write from host to device (a)synchronously (non-blocking).

```
cl_int clEnqueueReadBuffer( cl_command_queue command_queue,
                            cl_mem buffer, cl_bool blocking_read,
                            size_t offset, size_t cb,
                            void *ptr, cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list, cl_event *event )

cl_int clEnqueueWriteBuffer( cl_command_queue command_queue,
                             cl_mem buffer, cl_bool blocking_write,
                             size_t offset, size_t cb,
                             const void *ptr, cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list, cl_event *event )
```

  - Set blocking_[read|write] to CL_TRUE to wait until transfer completes. Use CL_FALSE for call to return immediately.
  - Host can carry on working while transfer occurs (don't use mem)
  - Wait on an *event* to be sure transfer has completed

Copying data via Command Queue

- Example copying to device via separate alloc and transfer

```
long            count;          // Example number of elements in my array
float           *vecA_h;        // Often use a _h suffix for host mem
cl_mem          vecA_d;         // Often use a _d suffix for device mem
cl_event        event;          // Indicates when transfer has completed

// Alloc host array and populate it (somehow)
vecA_h = (float *)malloc(count*sizeof(float));
readMyDataFromSomewhere(vecA_h, count);

// Create a read-only buffer: kernel will only read it, not write it
vecA_d = clCreateBuffer( ctx, CL_MEM_READ_ONLY, count*sizeof(float), NULL, &err );

// Enqueue a non-blocking write from host to device
err = clEnqueueWriteBuffer( cmd_queue, vecA_d, CL_FALSE, 0, count*sizeof(float),
                            vecA_h, 0, NULL, &event );
// Host can carry on with other work (don't touch vecA_h though!)
// E.g., create and populate another host array
...
// Eventually wait until previous transfer has completed
clWaitForEvents( 1, &event );

// Can now safely do something to the host array
free(vecA_h);
```

10. Set up Kernel Args

- Use a function to set our kernel inputs/outputs
  - Sadly we don't just call my_kernel(arg1, arg2, …);

```
cl_int clSetKernelArg( cl_kernel kernel,    cl_uint arg_index,
                       size_t arg_size,     const void *arg_value )
```

```
// Extract kernel from program earlier
// Allocated device memory objects earlier via clCreateBuffer() earlier

// Set the kernel args (refer your kernel source code!) Note err flag trick.
err  = clSetKernelArg( kernel, 0, sizeof(cl_int), &n );
err |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &vecA_d ); // Input mem obj
err |= clSetKernelArg( kernel, 2, sizeof(cl_mem), &vecB_d ); // Input mem obj
err |= clSetKernelArg( kernel, 3, sizeof(cl_mem), &vecC_d ); // Output mem obj
if ( err != CL_SUCCESS ) ...
```

```
__kernel void arrayAddOCL( const int n, __global const float *x,
                           __global const float *y, __global float *z )
{
  int i = get_global_id(0);
  z[i] = x[i] + y[i];
}
```

11. Launching the Kernel

- Add kernel launch to command queue
  - Think back to the NDRange layout of work-items (threads)
  - A non-blocking (async) function – returns immediately to host code

```
cl_int clEnqueueNDRangeKernel( cl_command_queue command_queue, cl_kernel kernel,
                               cl_uint work_dim,
                               const size_t *global_work_offset,
                               const size_t *global_work_size,
                               const size_t *local_work_size,
                               cl_uint num_events_in_wait_list,
                               const cl_event *event_wait_list, cl_event *event )
```

```
// 1-D example: Do the vector add for 64000 element arrays
int globalSize = 64000;          // Number of elements in our arrays
int localSize  = 128;            // Work-groups contain 128 work-items (500 wg's)

// No work offset (work-item ids start at 0). No events to wait for or returned.
err = clEnqueueNDRangeKernel( cmd_queue, kernel, 1, NULL, &globalSize
                              &localSize, 0, NULL, NULL );

// Command returns immediately to host, which can carry on working (see later)
```

Launching the Kernel

- The **work_dim** arg specifies array sizes for **work_**xx args
- global_work_offset changes initial value of get_global_id()
  - Pass in NULL to keep at 0 (,0, 0) (OpenCL 1.1)
- global_work_size gives total number of work-items reqd
- local_work_size gives work-group size

```
// 2-D example: Processing a matrix of size 2048 rows x 4096 columns
// Number of columns is 4096. This is the X dimension (or width)
// Number of rows is 2048. This is the Y dimension (or height)

int globalSize[2], localSize[2]; // work_dim is 2 so all args need two dimension
globalSize[0] = 4096;            // Dimension [0] is x (num cols)
globalSize[1] = 2048;            // Dimension [1] is y (num rows)
localSize[0]  = 32;              // 32x32 = 1024 work-items in a work-group
localSize[1]  = 32;              // Hence there will be 128x64 = 8192 work-groups

// No work offset (work-item ids start at 0). No events to wait for or returned.
err = clEnqueueNDRangeKernel( cmd_queue, kernel, 2, NULL, globalSize
                              localSize, 0, NULL, NULL );
```

Number of work-items and work-groups

- OpenCL calculates number of work-groups using
  - global_work_size[0] / local_work_size[0]
    global_work_size[1] / local_work_size[1] …
  - Your global_work_size numbers **must** be evenly divisible by your local_work_size numbers
    - err return flag will be CL_INVALID_WORK_GROUP_SIZE if not
    - Lots of other error flag values indicating where dimension info is wrong
- Your numbers must be within the hardware limits
  - Individual limits ([0], [1], …) and total limits ([0] x [1] x …)
  - clGetDeviceInfo() and ask for
    CL_DEVICE_MAX_WORK_ITEM_SIZES,
    CL_DEVICE_MAX_WORK_GROUP_SIZE etc

Number of work-groups

- Will need to experiment per device
  - Num work-groups > Num Compute Units (e.g., Nvidia SM)
    - *Each CU has at least one work-group to execute*
  - Num work-groups / Num Compute Units > 2
    - *Each CU can run multiple work-groups*
    - *Allows swapping of work-groups if one stalls (e.g., on a barrier – see later)*
  - Num work-groups / Num Compute Units > 100 (Nvidia recommendation)
    - *Your code will scale on future devices (won't run out of work)*

More work-items than needed

- What if the global / local numbers don't divide?
  - Silly example: your array length is prime
  - More realistic: global_work_size[0] =10000
    local_work_size[0] = 64
    - *Asking OpenCL for 156.25 work-groups!*
- Specify more work-items than reqd

```
int num_work_groups = (global_work_size[0]+local_work_size[0]-1)/local_work_size[0];
global_work_size[0] = num_work_groups*local_work_size[0];
```

  - Set global_work_size[0] = 10048
  - Kernel must check for going out of bounds in your arrays

```
__kernel void arrayAddOCL( const int n, __global const float *x,
                           __global const float *y, __global float *z )
{
  int i = get_global_id(0);          // 10048 work-items hence i could = 10000+
  if ( i < n )                       // Pass in n=10000 i.e., the true array len
    z[i] = x[i] + y[i];
}
```

Choosing the work-group size

- local_work_size arg can be NULL
  - OpenCL will choose the "best" value
  - Don't rely on it being the "best"
  - Often recommended to be multiple of warp/wavefront size
    - *Nvidia suggest multiple warps per work-group*
- You'll need to experiment to find the best value
  - Run kernel with several values and time it (more later)
  - Can get the value the device thinks is the "best"

```
// Get device-specific kernel info
size_t max_wg_size;     // Your local_work_size[0] x local_work_size[1] ...
                        // should equal this value.

err = clGetKernelWorkGroupInfo( kernel, device_id, CL_KERNEL_WORK_GROUP_SIZE,
                                sizeof(size_t), &max_wg_size, NULL );
```

12. Transfer results back to Host

- Enqueue a read command to get data back from device
  - As before, transfer can be blocking or non-blocking
  - Don't transfer back if simply passing cl_mem obj to another kernel
  - An in-order queue guarantees the kernel has finished

```
long          count;          // Example number of elements in my array
float         *vecC_h;        // Host array to be populated with results
cl_mem        vecC_d;         // Device mem obj written to by kernel

// Kernel was Enqueued earlier and may be running...

// Alloc host array to hold results from device (usually allocated earlier)
vecC_h = (float *)malloc(count*sizeof(float));

// Enqueue a blocking read from device to host
err = clEnqueueReadBuffer( cmd_queue, vecC_d, CL_TRUE, 0, count*sizeof(float),
                           vecC_h, 0, NULL, NULL );

// The above call will not return until transfer is complete
writeMyArrayToFile(vecC_h, count);
```

Waiting for the Kernel

- How do we know the kernel has finished?
  - In-order queue: the clEnqueueReadBuffer() command won't run until previous commands in queue have completed
  - Out-of-order queue: Need to wait on an event or barrier (see later)

```
// 1-D example: Do the vector add for 64000 element arrays
int globalSize = 64000;          // Number of elements in our arrays
int localSize  = 128;            // Work-groups contain 128 work-items (500 wg's)
cl_event event;                  // Indicate when kernel has finished

// No work offset. No events to wait for. An event is written to when complete.
err = clEnqueueNDRangeKernel( cmd_queue, kernel, 1, NULL, &globalSize
                              &localSize, 0, NULL, &event );

// Command returns immediately to host, which can carry on working
...
// Wait for one event then transfer data back from device to host (blocking call)
err = clEnqueueReadBuffer( cmd_queue, vecC_d, CL_TRUE, 0, count*sizeof(float),
                           vecC_h, 1, &event, NULL );
```

13. Tidying Up

- Free resources on device as you would on host
  - Free up device memory (buffers, objects etc)
    - *At end of program and between kernels if no longer required*
    - *'Creation' commands have a 'release' equivalent*
    - *Typically call in reverse order (although reference counting is used)*

```
// This ensures all previous commands have finished (may not be needed)
clFinish( cmd_queue );

// Release device objects
clReleaseMemObject( vecA_d );
clReleaseMemObject( vecB_d );
clReleaseMemObject( vecC_d );
clReleaseProgram( prog );
clReleaseKernel( kernel );
clReleaseCommandQueue( cmd_queue );
clReleaseContext( ctx );

// Don't forget host objects - let's be tidy
free( vecC_h );
```

Steps 1-13 Done

- Sample app. No error checking (not recommended!)

```
#include <CL/opencl.h>

// Kernel as one long single string
const char *ksource = \
"__kernel void myKernel( const int n, __global float *inArr, __global float *outArr ) { \n" \
"  int idx = get_global_id(0);\n " \
"  // Do something wonderful in OpenCL \n" \
"  outArr[i] = inArr[i]*inArr[i]; \n" \
"}\n";

#define VSIZE 1024*1024
int main( void )
{
  cl_int                  err;
  cl_platform_id          p_id;
  cl_device_id            d_id;
  cl_context_properties cprops = {CL_CONTEXT_PLATFORM,0,0};
  cl_command_queue        cq;
  cl_program              prog;
  cl_kernel               kernel;
  cl_mem                  A_d, B_d;
  float                   A_h[VSIZE], B_h[VSIZE];
  long                    nbytes;
  int                     vsize = VSIZE;

  nbytes = VSIZE*sizeof(float);          // Num bytes to alloc
  initArray( A_h, VSIZE);                // Random values?

  // Assuming just one platform, one GPU
  err = clGetPlatformIDs( 1, &p_id, &num_platforms );
  clGetDeviceIDs(p_id, CL_DEVICE_TYPE_GPU, 1, &d_id, NULL);
  cprops[1] = p_id;
  ctx = clCreateContext(cprops, 1, &d_id, NULL, NULL, NULL);
  cq = clCreateCommandQueue(ctx, d_id, 0, NULL);
  prog = clCreateProgramWithSource(ctx, 1, &ksource, NULL, NULL);
  clBuildProgram( prog, 0, NULL, NULL, NULL, NULL );

  // Zeros at end of functions should really be NULLs

  kernel = clCreateKernel(prog, "myKernel", NULL);
  A_d = clCreateBuffer(ctx, CL_MEM_READ_ONLY, nbytes, 0, 0);
  B_d = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY, nbytes, 0, 0);

  // Enqueue a blocking write from host to device mem obj
  clEnqueueWriteBuffer(cq, A_d, CL_TRUE, 0, nbytes, A_h, 0, 0,0);

  // Set the kernel args
  clSetKernelArg(kernel, 0, sizeof(int),    &vsize);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &A_d);
  clSetKernelArg(kernel, 2, sizeof(cl_mem), &B_d);

  // Enqueue a kernel launch using a 1-D NDRange. Auto W-G size.
  clEnqueueNDRangeKernel(cq, kernel, 1, NULL, &vsize, 0, 0,0,0);

  // Enqueue a blocking readback from the device
  clEnqueueReadBuffer(cq, B_d, CL_TRUE, 0, nbytes, B_h, 0, 0, 0);

  // Could do something with the results in B_h now ...

  // Tidy up
  clReleaseMemObject( A_d );
  clReleaseMemObject( B_d );
  clReleaseProgram( prog );
  clReleaseKernel( kernel );
  clReleaseCommandQueue( cq );
  clReleaseContext( ctx );
}
```

MANCHESTER 1824

The University of Manchester

OpenCL C and Kernels

OpenCL Kernels

- Written in OpenCL C
  - Restricted version of ISO C99
    - *No recursion, function pointers, or functions from the standard headers*
  - Preprocessor (C99) supported
    - *#ifdef etc*
  - Scalar datatypes plus vector types
    - *E.g., float4, int16 etc (with host equivalents: cl_float4, cl_int16)*
  - Image types: image2d_t, image3d_t etc
    - *Alternative to memory buffers for images (not covered)*
  - Built-in functions: already seen work-item/group id functions
    - *Many math.h functions, synchronisation functions*
    - *Optional functions up to vendors: e.g., double precision*

OpenCL Kernels

- Remember: one instance of the kernel created for each work-item (thread)

- Must have __kernel keyword
  - otherwise is a function only a kernel can call
- No return value allowed (void)
- Declare address space of memory object args

```
__kernel void simpleOCL(const int n, __global float *x )
{
  int i = get_global_id(0);
  if ( i < n )
    x[i] = sin( (float)i );
}
```

- Lots of builtin maths / geometry functions
- Work out which data a work-item uses
  - get_global_id(0), get_local_id(1) etc
  - 2D: column = x = get_global_id(0), row = y = get_global_id(1)

1-D Example

- Raise all elements of a vector by the given exponent
  - 1-D index-space

```
__kernel void exponentor(__global int* data, const uint exponent)
{
  int tid = get_global_id(0);
  int base = data[tid];
  int i;
  for (i=1; i < exponent; ++i) {
    data[tid] *= base;
  }
}
```

- See trick later to avoid passing in the exponent param

Example by Intel: http://software.intel.com/en-us/articles/tips-and-tricks-for-kernel-development/

2-D Example

- Rotate a 2-D image about its centre
  - OpenCL has built-in image types and functions. For this simple example we use an ordinary memory buffer.
  - WARNING: there is a horrible inefficiency in this kernel. Spotted it?

Host code

```
// We should be launched with a 2-D index space. For example:
int globalSize[2] = {2048,2048};   // Input 'image' array is 2k x 2k values
int localSize[2]  = {64,64};       // Divides evenly the globalSize[]
```

```
__kernel void rotArray( const float theta, const int w, const int h,
                        __global const int *inImage, __global int *outImage)
{
  int dstx = get_global_id(0);     // This work-item's point in index-space
  int dsty = get_global_id(1);     // is where we will write to (not read from)

  // We read from elsewhere in the image data
  int srcx = (int)floor( ((float)dstx)*cos(theta) + ((float)dsty)*sin(theta) );
  int srcy = (int)floor( ((float)dsty)*cos(theta) - ((float)dstx)*sin(theta) );

  // Check bounds before reading from elsewhere
  if ( (srcx>=0 && srcx<w) && (srcy>=0 && srcy<h) ) {
    // Can access memory buffers using 1-D indexing or 2-D (as with C)
    outImage[dsty*w+dstx] = inImage[srcy*w+srcx];
  }
}
```

Based on example by Perhaad Mistry, Daba Schaa, Benedict Gaster, AMD OpenCL University Kit

MANCHESTER 1824

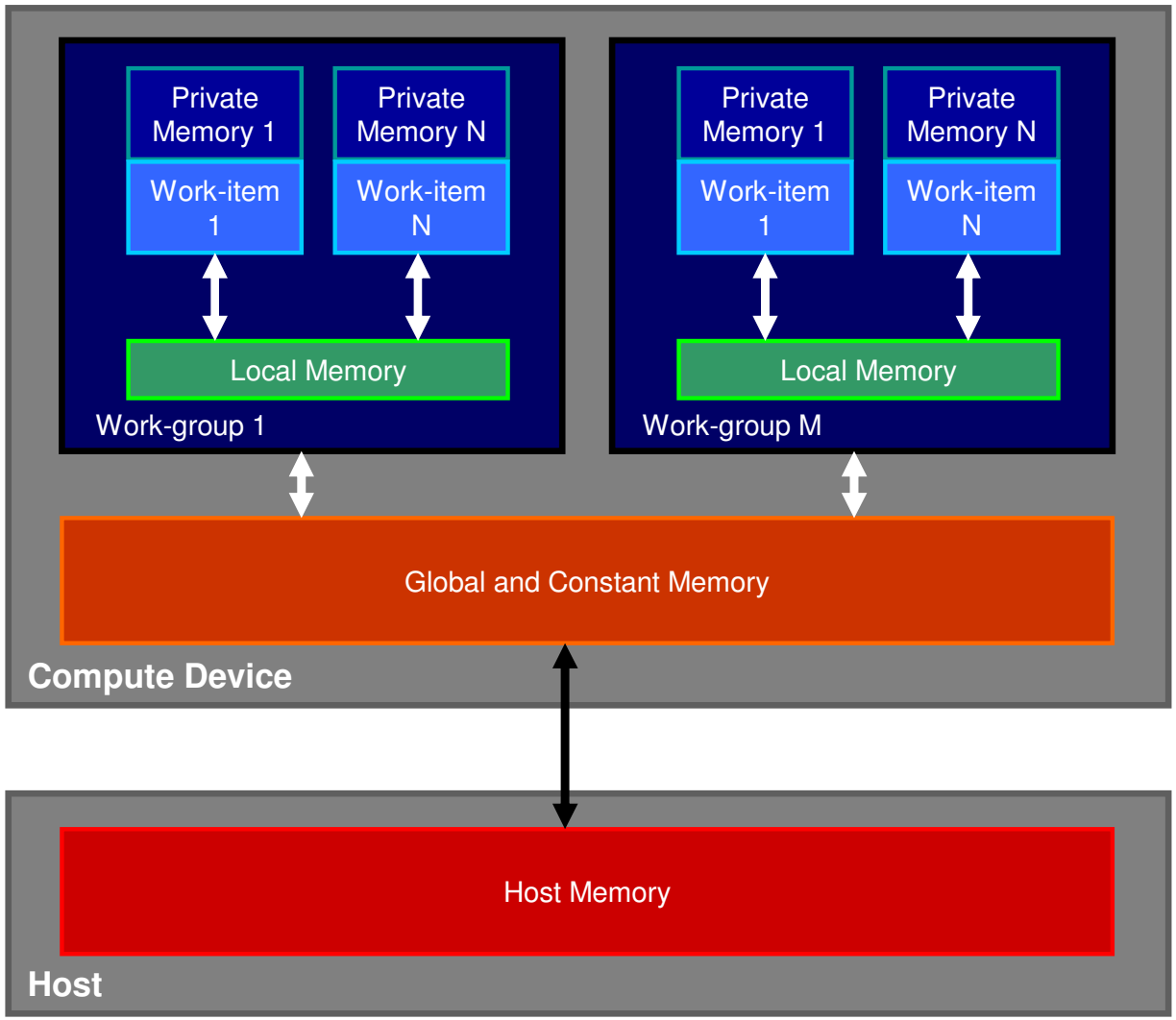
The University of Manchester

Memory and Kernels

Memory hierarchy

Kernel Address Space Qualifiers

- The OpenCL memory model
- Private mem: __private
  - R/W per work-item
- Local mem: __local
  - R/W by the work-items in a work-group
- Global mem: __global
  - R/W by work-items in all work-groups
- Const mem: __constant
  - R-only global mem visible to all work-items
    Host allocs and inits
- Host mem:
  - On the CPU



- Explicit memory management: You move data host->global->local and back

Private Memory

- A kernel's local variables and a function's args are private
  - E.g., **x, y, n, a, b, temp, p, i** all automatically __private to each work-item

```
float myFunc( float x, float y ) {
  return 2.0*x*x*x+3.0*y*y-1.0;
}
__kernel void myKernel( const int n, ... ) {
  float a, b;
  float temp[4];
 __global int *p;
 int i = get_global_id(0);
 // Kernel body will call myFunc(a,b) at some point
}
```

- GPUs alloc __privates in Compute Unit's register file
  - Fastest access times
  - Limited number of registers per CU (and hence per work-item)
  - More registers implies fewer work-groups executing concurrently

Local Memory

- Use to share items between work-items in a work-group
  - Allows work-items to communicate (within their work-group)
  - Can save re-loading data from global memory
  - Can replace a __private (register) if same value in all work-items
  - Useful for (temp) arrays – quicker than using global memory

```
__kernel void myKernel( const int n, ... )
{
  __local float tmpScale;           // Cannot initialize here
  __local float wgValues[64];       // Hard-coded size in kernel
  int g_idx = get_global_id(0);     // global work-item id
  int l_idx = get_local_id(0);      // local work-item id in work-group
  tmpScale = 123.456;               // Bad: All work-items do this! Use __constant?

  wgValues[l_idx] = myComputation(g_idx, ...);
  ...
}
```

Memory Consistency

- Local memory highlights a potential problem
  - All work-items in a work-group writing to a local array
  - How do we know when all work-items have written (or read, etc)?

```
__local float wgValues[64];          // Assume work-group size of 64
wgValues[get_local_id(0)] = myComputation(...);
// Have all work-items in work-group written to wgValues[]? See later
if ( get_local_id(0) == 0 )          // Lots of idle work-items!
  for ( i=0; i<64; i++ )
    sum += wgValues[i];
```

- Memory model has *relaxed consistency*
  - Different work-items may see a different view of local and global memory as computation progresses
  - Load/store consistency within a work-item
  - Global & local mem consistent within a work-group at barrier
  - Command queue events ensure consistency between kernel calls

Local Memory cont…

- Do we have to hard-code a __local mem array size?
  - E.g., Often want size equal to number of work-items in work-group
  - Hard-coding the size is not portable in this case
  - No dynamic mem alloc (e.g., malloc) in a kernel
  - Could use a preprocessor value (reqs kernel recompiles)
- Pass size as kernel arg from host (in a slightly odd way)
  - Setting last arg of clSetKernelArg() to NULL means __local mem

```
err |= clSetKernelArg( kernel, 0, sizeof(int),    &numElems ); // As earlier
err |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &vecA_d );   // As earlier
err |= clSetKernelArg( kernel, 2, sizeof(cl_float)*localSize, NULL );
```

```
__kernel void myKernel(const int n, __global float *x, __local float *localArray)
{
  int local_idx = get_local_id(0);          // 0 ... get_local_size(0)-1
  localArray[local_idx] = ...;              // work-items in work-group init array
}
```

Global Memory

- Used to refer to memory objects (buffers or images)
  - Memory allocated with clCreateBuffer() is in global memory
  - Usually passed in to kernel as an arg
  - Can pass to functions

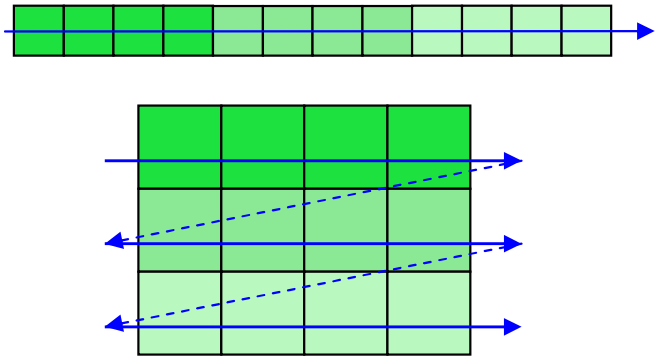
```
float myFunc( int i, __global const float *x ) {
  float tmp = *(x+i);                  // Could also use x[i]
  return 2.0*tmp*tmp*tmp+3.0*sin(tmp)-1.0;
}

__kernel void myKernel( const int n,
                        __global const float *inArr,
                        __global float *outArr )
{
  int i = get_global_id(0);
  outArr[i] = myFunc(i, inArr);
}
```

Accessing Memory

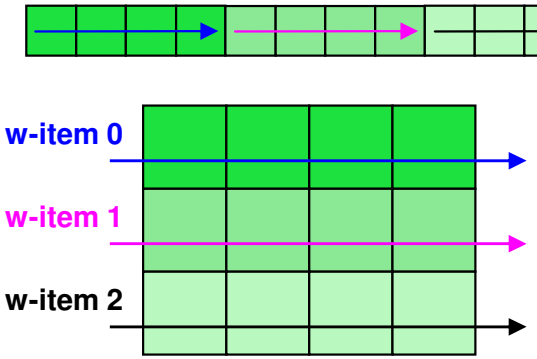
- Coalesced __global memory access for efficiency
  - work-item i to access array[i]
    work-item i+1 to access array[i+1]
- Access to C matrices (row-major)

  - E.g., a 1D index-space where a single work-item processes an *entire row* of a C matrix (seen in mat-mul examples)

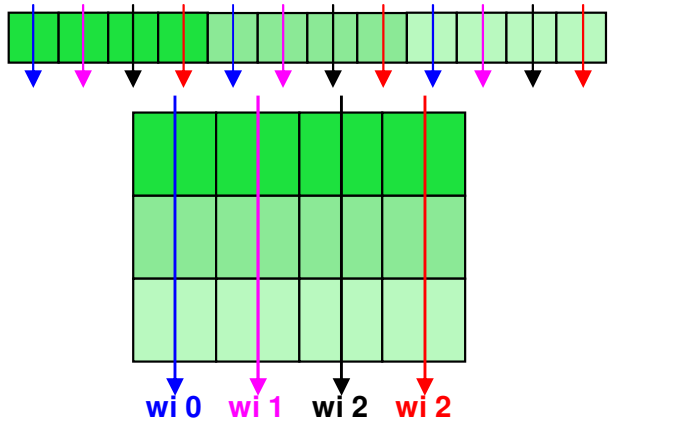
```
// C (CPU cache-friendly)
for (r=0; r<height; r++)
  for (c=0; c<width; c++)
    z = A[r*width+c];
```



```
// Kernel (non-coalesced)
WI=get_global_id(0);
for (c=0; c<width; c++ )
  z = A[WI*width+c];
```




```
// Kernel (coalesced)
WI=get_global_id(0);
for (r=0; r<height; r++ )
  z = A[r*width+WI];
```



MANCHESTER 1824

The University of Manchester

Synchronization

In-kernel barriers, queue barriers, events, timing

Synchronization

Various synchronization points (seen some already)

- Between work-items in a work-group
- Between kernels (and other commands) in a queue
- Between kernels (and other commands) in separate queues
- Between the host and the queues

- Sync-ing can be enforced/implied by OpenCL
  - An in-order queue: commands executed in order submitted
- Or explicitly requested by user
  - Out-of-order queue: commands scheduled by OpenCL
  - Barriers in kernels and queues
  - Events in queues

Work-item Synchronization

- Only possible within a work-group
  - Can't sync with work-items in other work-groups
  - Can't sync one work-group with another
- Use **`barrier(`*`type`*`)`** in kernel where *type* is
  - **`CLK_LOCAL_MEM_FENCE`**: ensure consistency in local mem
  - **`CLK_GLOBAL_MEM_FENCE`**: ensure consistency in global mem
- All work-items in work-group must issue the barrier() call and same number of calls

```
__kernel void BadKernel(...) {
  int i = get_global_id(0);
  ...
  // ERROR: Not all WIs reach barrier
  if ( i % 2 )
    barrier(CLK_GLOBAL_MEM_FENCE);
}
```

```
__kernel void BadKernel(...) {
  int i = get_local_id(0);
  ...
  // ERROR: WIs issue different number
  for ( j=0; j<=i; j++ )
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Use with __local memory

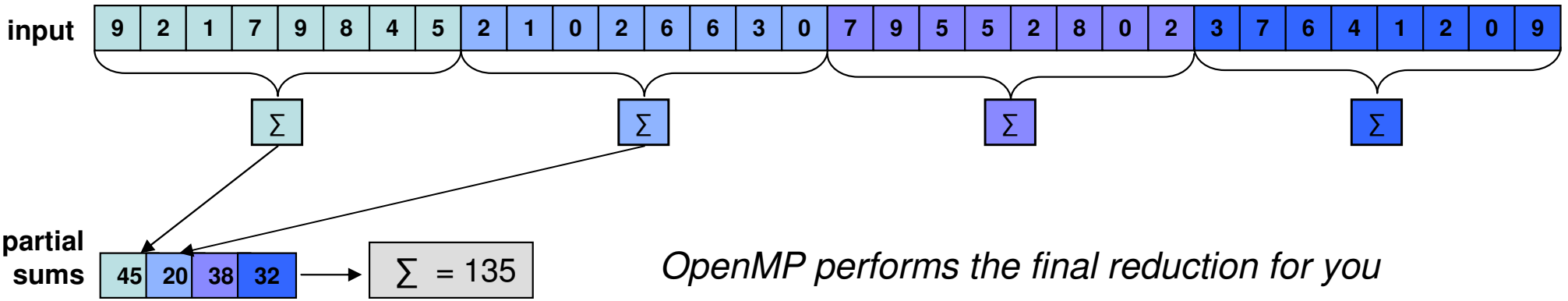
- Barrier often used when initializing __local memory
  - Kernel must initialize local memory

```
__kernel void kMat( const int n, __global float *A, __local float *tmp_arr )
{
  // Could also have some fixed size local array
  // __local float tmp_arr[64];

  int gbl_id = get_global_id(0);   // ID within entire index space
  int loc_id = get_local_id(0);    // ID within this work-group
  int loc_sz = get_local_size(0);  // Size of this work-group

  // For some reason we want to fill up the first half of the local array
  if ( loc_id < loc_sz/2 )
    tmp_arr[loc_id] = A[gbl_id];

  // All work-items must hit barrier. They'll all see a consistent tmp_arr[]
  barrier(CLK_LOCAL_MEM_FENCE);

  // Each work-item can now use the elements from tmp_arr[] safely.
  // Often used if we'd be repeatedly accessing the same A[] elements.
  for ( j=0; j<loc_sz; j++ )
    my_compute( gbl_id, tmp_arr[j], A );
}
```

Example: Parallel Reduction (CPU)

- Reduction of an array of numbers to a single value
  - E.g. sum:
  - OpenMP on host forms partial sums in each thread
    - *Linear (serial) sum within thread*

```
void sumCPU( const int n, const float *x,
             float *res )
{
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for ( int i=0; i<n; i++ )
    sum += x[i];
  *res = sum;
}
```

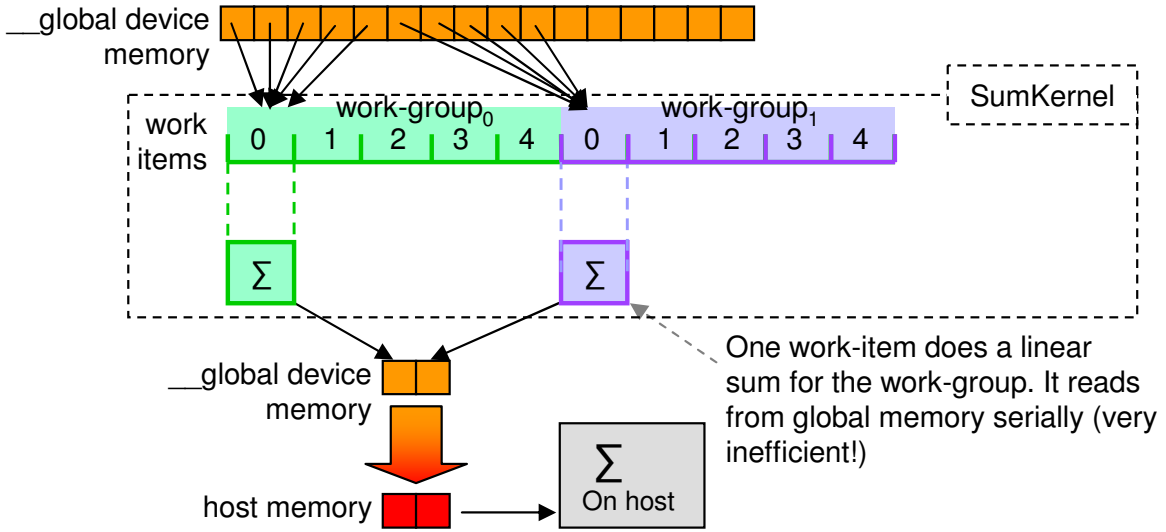


OpenMP performs the final reduction for you

- Can recreate in OpenCL using work-groups

Parallel Sum on GPU (1st attempt)

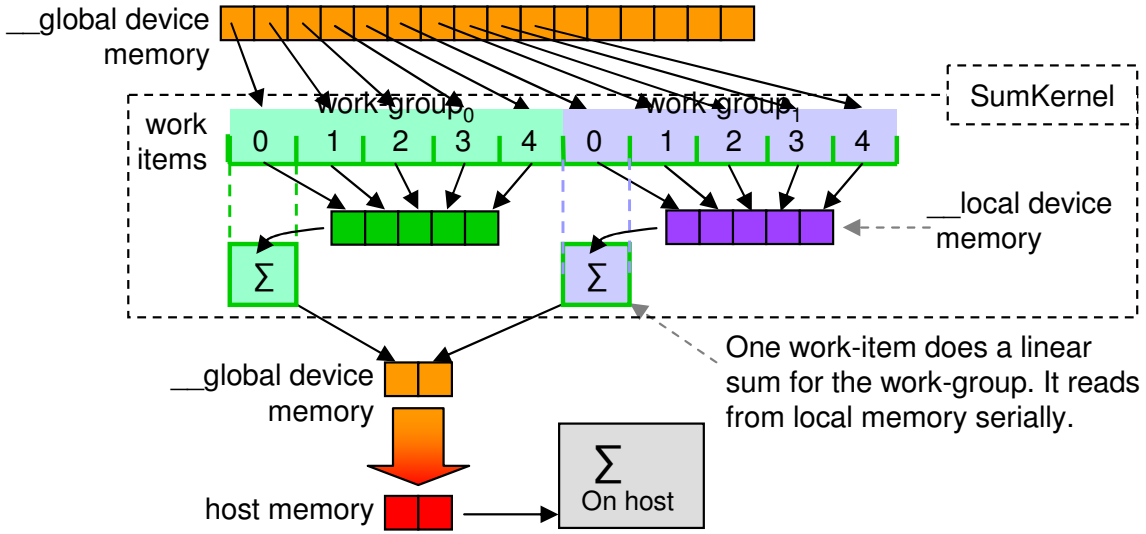
- Use one work-item to perform a sum within a work-group
  - Inefficient – only one work-item forms the partial sum




```
__kernel void sumGPU1( const uint n, __global const float *x,
                       __global float *partialSums ) {
  if ( get_local_id(0) == 0 ) {                    // Many idle work-items!
    float group_sum = x[get_global_id(0)];
    for ( int i=1; i<get_local_size(0); i++ )
      group_sum += x[get_global_id(0)+i];          // Should check (gid+i) < n
    partialSums[get_group_id(0)] = group_sum;      // Write sum to output array
  }
  // Add barrier(CLK_GLOBAL_MEM_FENCE) if doing other work in kernel
}
```

Parallel Sum on GPU (2nd attempt)

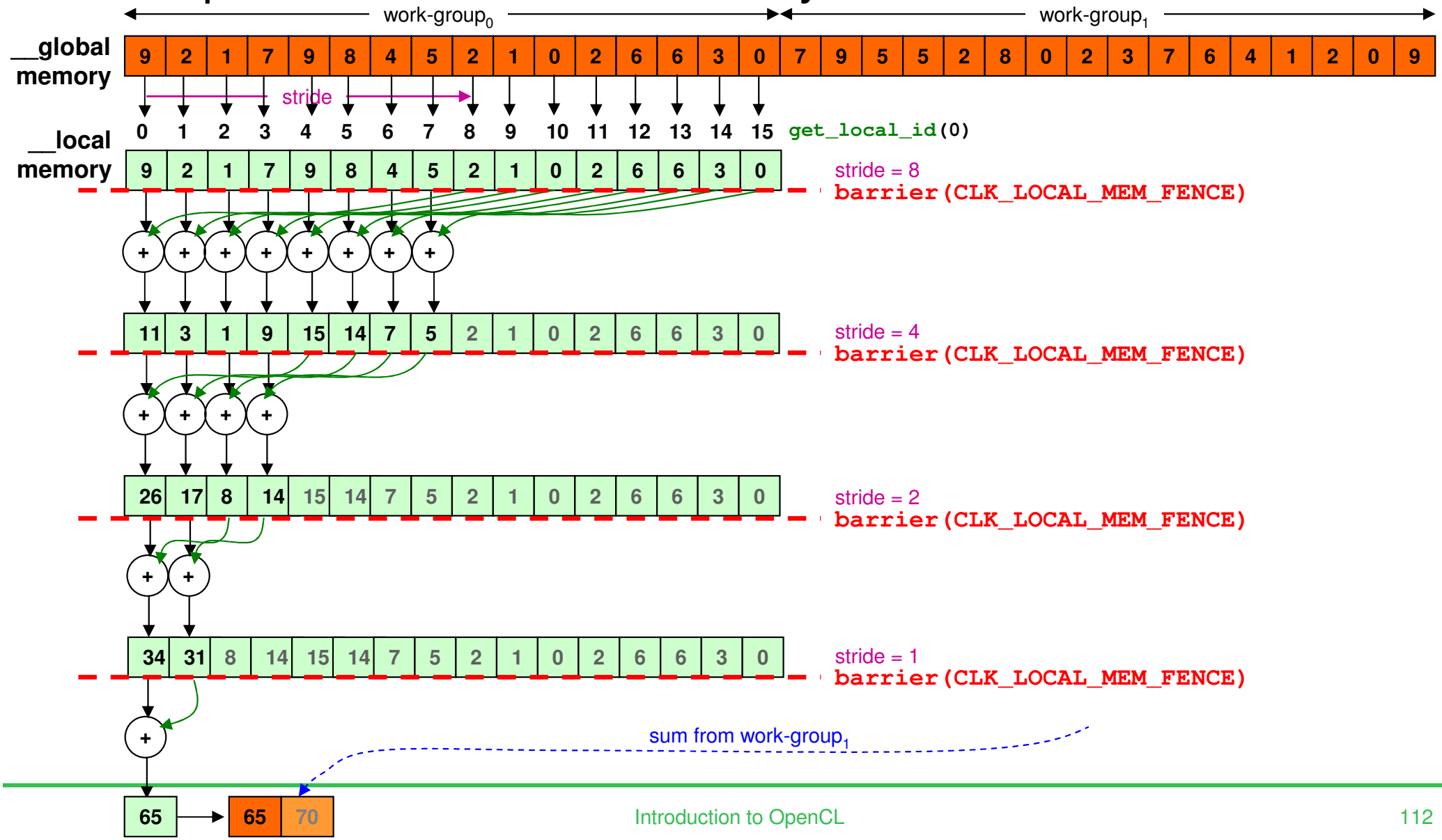
- Memory optimization – copy to __local memory in parallel
  - Still inefficient – only one work-item forms the partial sum




```
__kernel void sumGPU2( const uint n, __global const float *x,
                       __global float *partialSums, __local float *localCopy ) {
  localCopy[get_local_id(0)] = x[get_global_id(0)]; // Init the localCopy array
    barrier(CLK_LOCAL_MEM_FENCE);                   // All work-items must call
  if ( get_local_id(0) == 0 ) {                     // Many idle work-items!
    float group_sum = localCopy[0];
    for ( int i=1; i<get_local_size(0); i++ )
      group_sum += localCopy[i];                    // Sum up the local copy
    partialSums[get_group_id(0)] = group_sum;       // Write sum to output array
  } // Add barrier(CLK_GLOBAL_MEM_FENCE) if doing other work in kernel
}
```

Parallel Reduction (3rd attempt)

- Improve on linear sum – binary sum



Parallel Reduction (3rd attempt) Kernel

- Copy all work-group's values in to __local memory
  - Repeatedly half the work-group, adding one half to the other

```
__kernel void sumGPU3( const uint n, __global const float *x,
                       __global float *partialSums, __local float *localSums )
{
  uint local_id   = get_local_id(0);
  uint group_size = get_local_size(0);

  // Copy from global mem in to local memory (should check for out of bounds)
  localSums[local_id] = x[get_global_id(0)];
  for (uint stride=group_size/2; stride>0; stride /= 2) { // stride halved at loop

    // Synchronize all work-items so we know all writes to localSums have occurred
    barrier(CLK_LOCAL_MEM_FENCE);

    // First n work-items read from second n work-items (n=stride)
    if ( local_id < stride )
      localSums[local_id] += localSums[local_id + stride]
  }
  // Write result to nth position in global output array (n=work-group-id)
  if ( local_id == 0 )
    partialSum[get_group_id(0)] = localSums[0];
}
```

Improved Reduction (4th attempt) Kernel

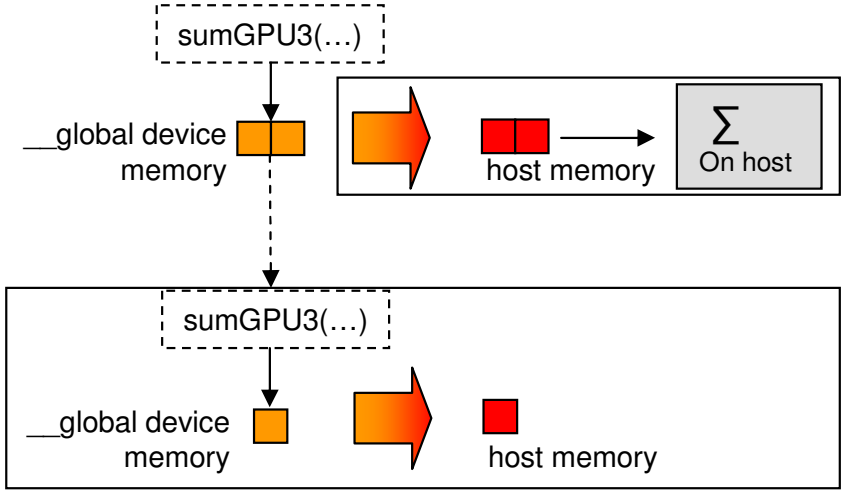
- Slight re-order to remove a couple of loop iterations
  - NB: set global_work_size to be half the input array length

```
__kernel void sumGPU4( const uint n, __global float *x,
                       __global float *partialSums, __local float *localSums ) {
  uint global_id   = get_global_id(0);     // Gives where to read from
  uint global_size = get_global_size(0);   // Used to calc where to read from
  uint local_id    = get_local_id(0);      // Gives where to read/write local mem
  uint group_size  = get_local_size(0);    // Used to calc initial stride

  // Copy from global mem in to local memory (doing first iteration)
  localSums[local_id] = x[global_id] + x[global_id + global_size];
  barrier(CLK_LOCAL_MEM_FENCE);
  for (uint stride=group_size/2; stride>1; stride>>=1 ) { // >>=1 does same as /=2
    // First n work-items read from second n work-items (n=stride)
    if ( local_id < stride )
      localSums[local_id] += localSums[local_id + stride];

    // Synchronize so we know all work-items have written to localSums
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  // Last iter: write result to nth position in global x array (n=work-group id)
  if ( local_id == 0 )
    x[get_group_id(0)] = localSums[0]+localSums[1];
}
```

Kernel based on slides by Tim Mattson, Intel, ISC2011

Partial Sums

- Still have *n* partial sums (*n*=number of work-groups)
  - Sum on host
  - Sum on GPU
    - *Linear sum (use one work-item)*
    - *Parallel reduction (use one work-group) provided n is small enough. Iterate if not.*



- Both GPU options can be done with another kernel call
  - Data is still on the GPU (in the partialSums array)
    - *Avoid a device-to-host transfer*
    - *Simply make another kernel call passing in the device memory object*
    - *DO NOT transfer back to host then pass back to device!*

What to read next

- For CSF (Nvidia), docs are available in

  `$CUDA_HOME/doc/`
  (don't forget **`module load libs/cuda/`**)

  `OpenCL_Programming_Overview.pdf`

  `OpenCL_Programming_Guide.pdf`

  `OpenCL_Best_Practices_Guide.pdf`

  – Do read the OpenCL specification available at

  http://www.khronos.org/registry/cl/

MANCHESTER 1824

The University of Manchester

Directorate of IT Services

Research Computing Services

http://www.manchester.ac.uk/researchcomputing
rcs@manchester.ac.uk