

OpenCL and CUDA: A Hands-on Introduction

Tim Mattson
Intel Corp.

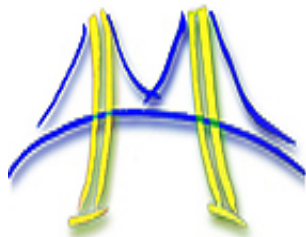
Acknowledgements: These slides are based on content produced by Tom Deakin and Simon McIntosh-Smith from the University of Bristol which were based on slides by Tim and Simon with Ben Gaster (Qualcomm) .



Disclaimer

READ THIS ... its very important

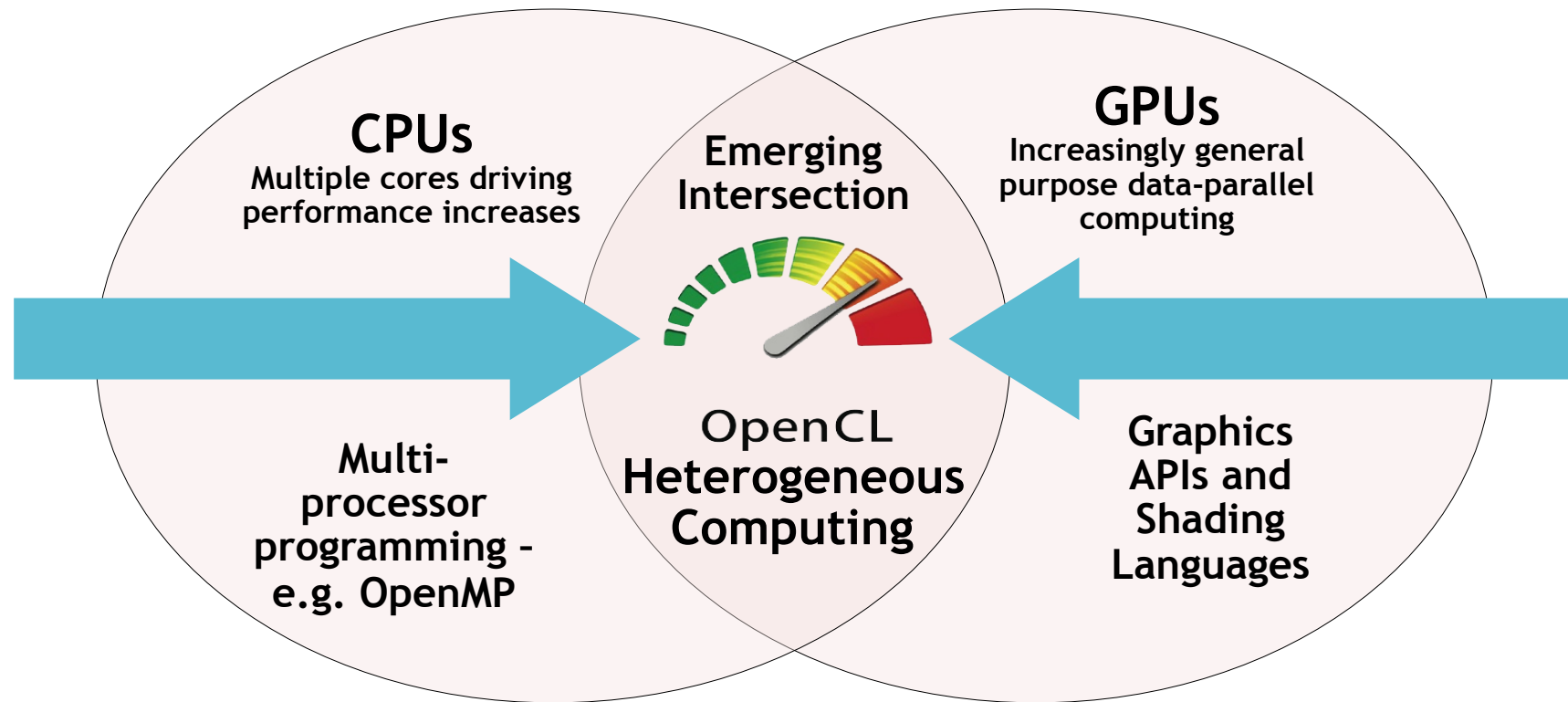
- The views expressed in this talk are those of the speakers and not their employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if we say anything really stupid, it's our fault ... don't blame our collaborators.



Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 ... A UC Berkeley course on Architecting parallel applications with Design Patterns.

AN INTRODUCTION TO OPENCL

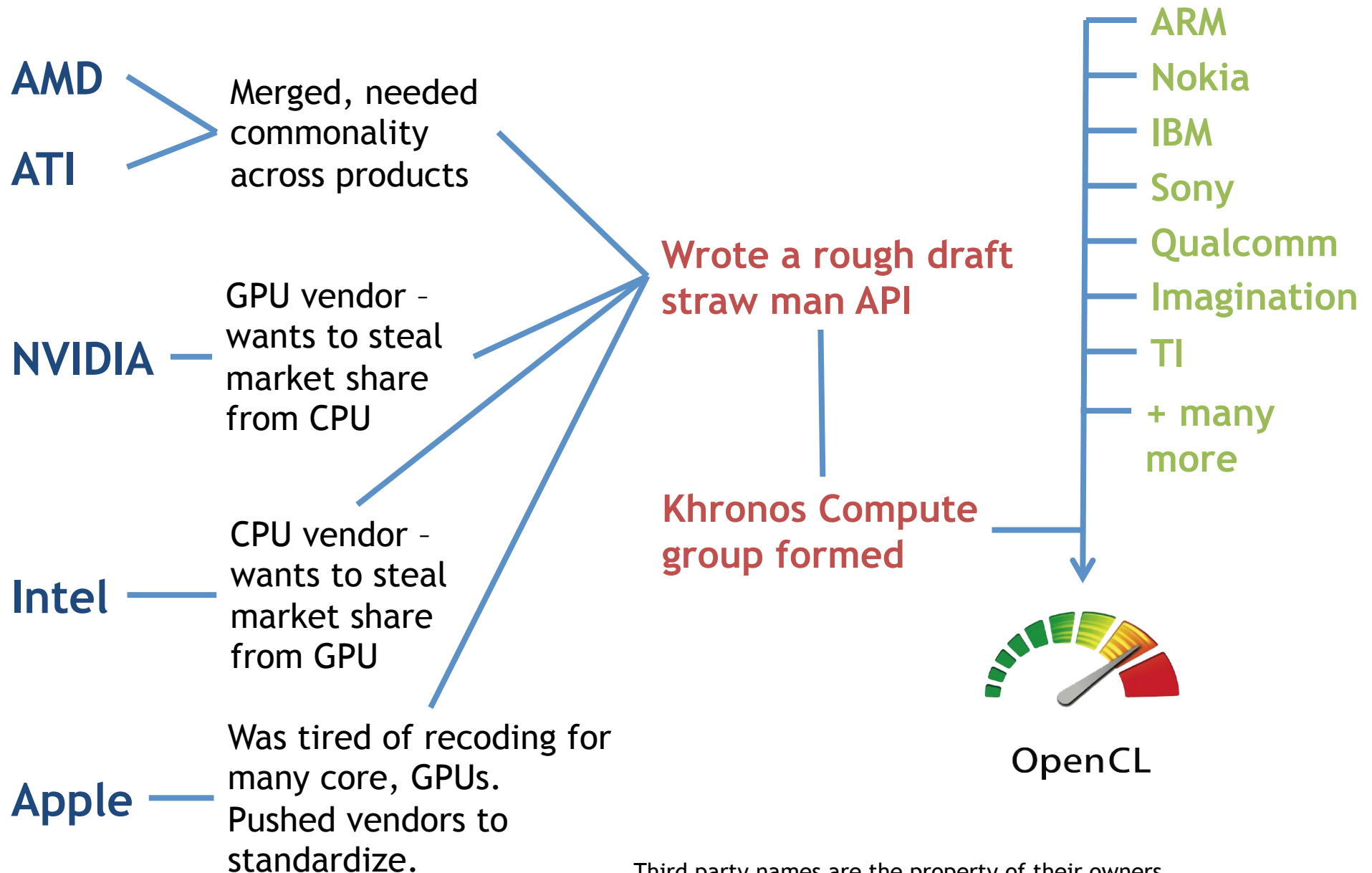
Industry Standards for Programming Heterogeneous Platforms



OpenCL - Open Computing Language

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

The origins of OpenCL



Third party names are the property of their owners.

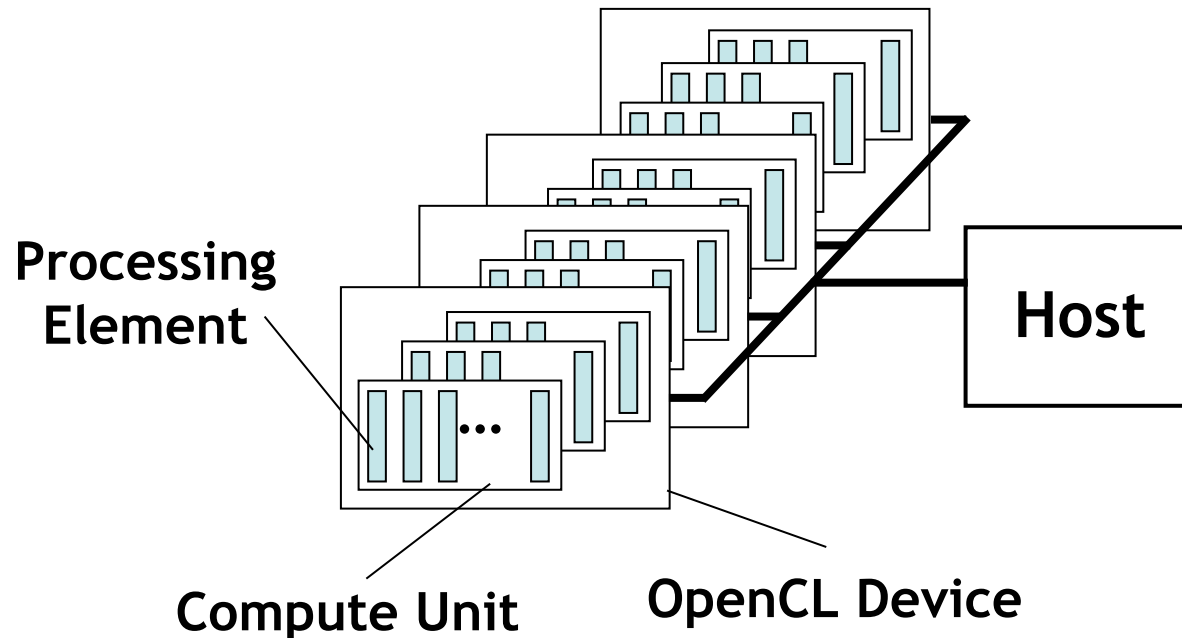
OpenCL: From cell phone to supercomputer

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



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

OpenCL Platform Model



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

The **BIG** idea behind OpenCL

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

Traditional loops

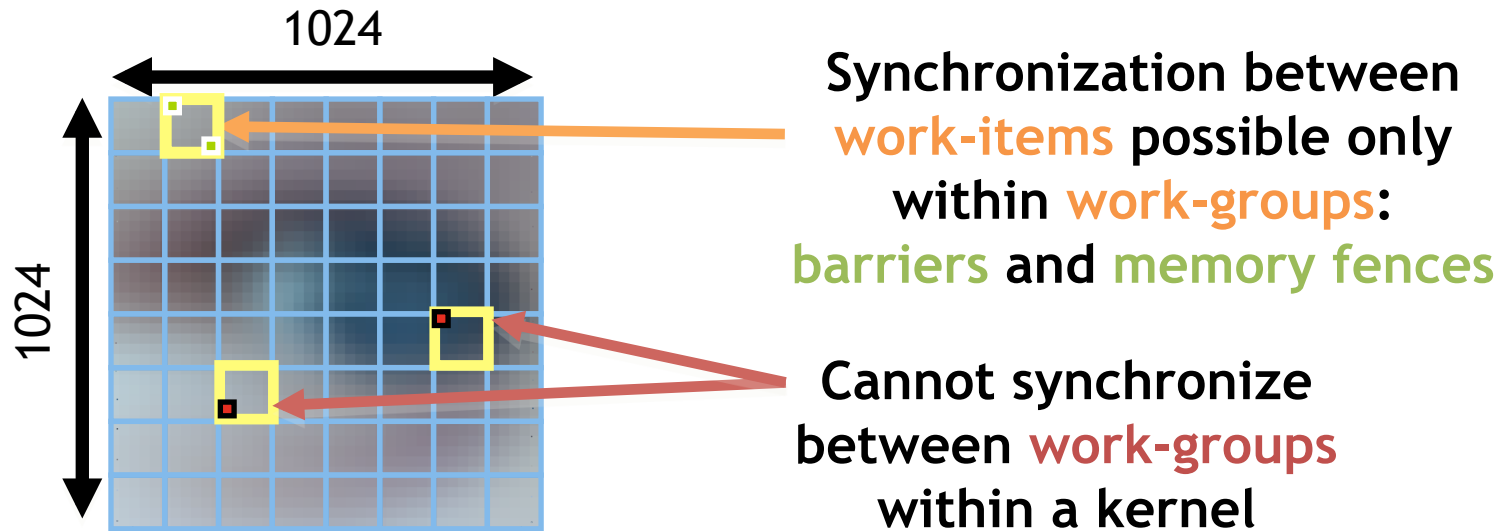
```
void
mul(const int n,
    const float *a,
    const float *b,
    float *c)
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
// execute over n work-items
```


An N-dimensional domain of work-items

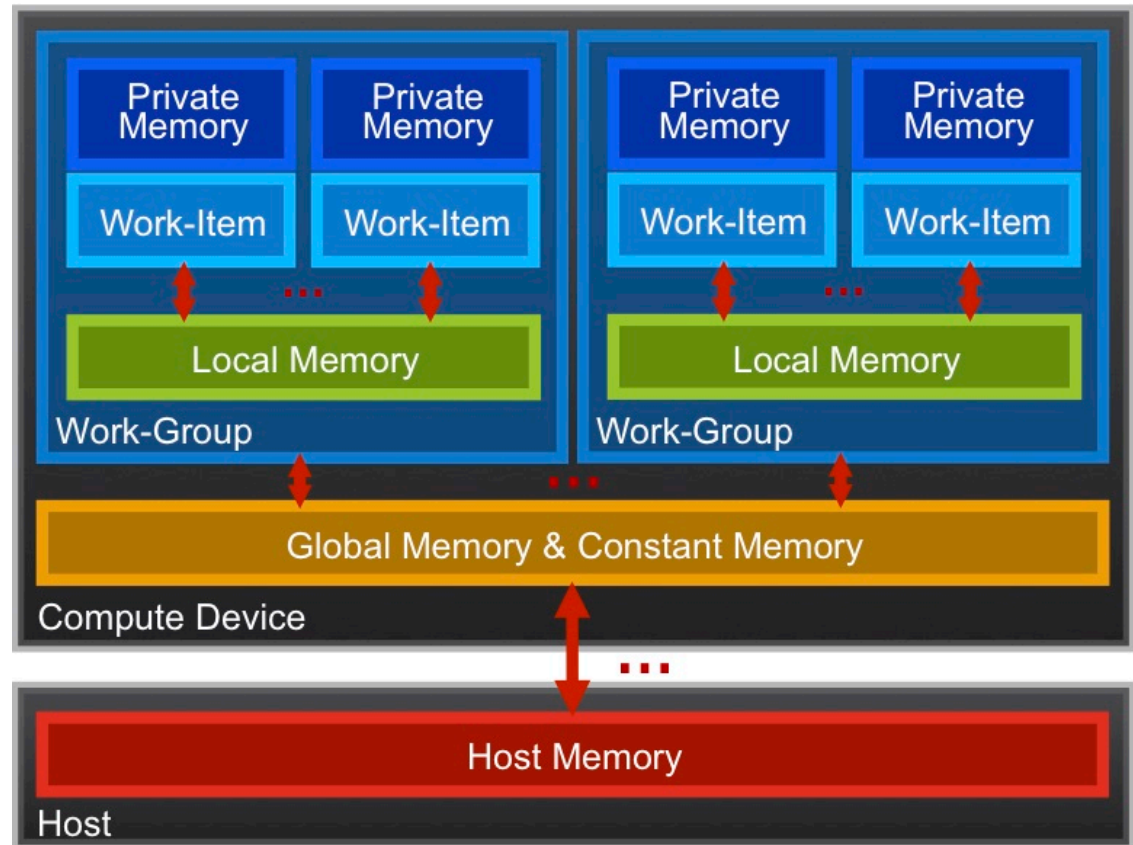
- **Global** Dimensions:
 - 1024x1024 (whole problem space)
- **Local** Dimensions:
 - 128x128 (**work-group**, executes together)



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

OpenCL Memory model

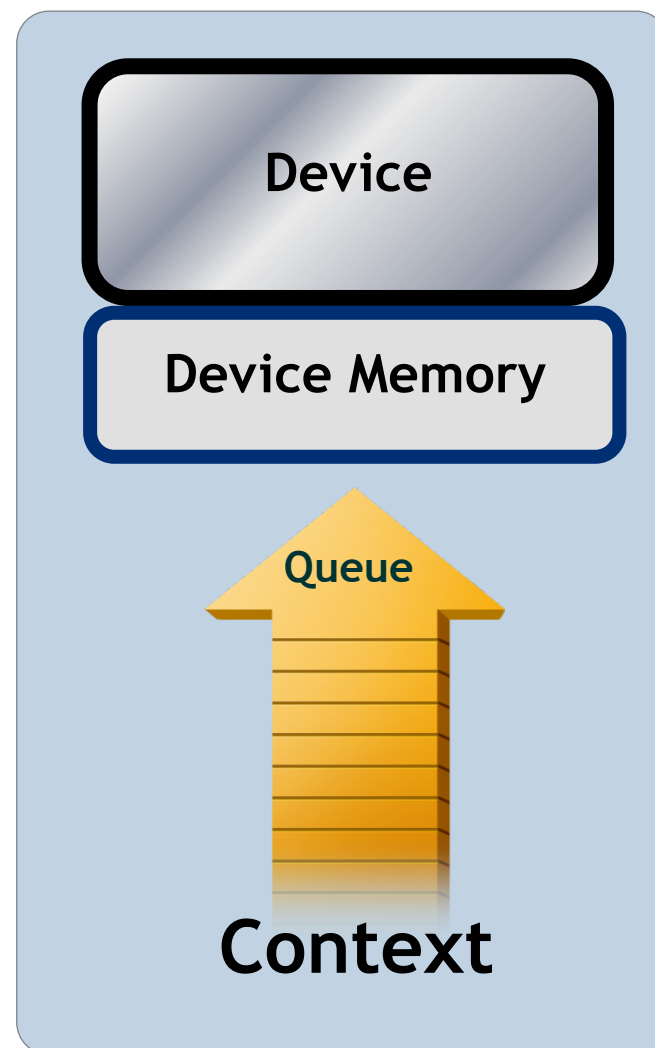
- *Private Memory*
 - Per work-item
- *Local Memory*
 - Shared within a work-group
- *Global Memory / Constant Memory*
 - Visible to all work-groups
- *Host memory*
 - On the CPU



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

Context and Command-Queues

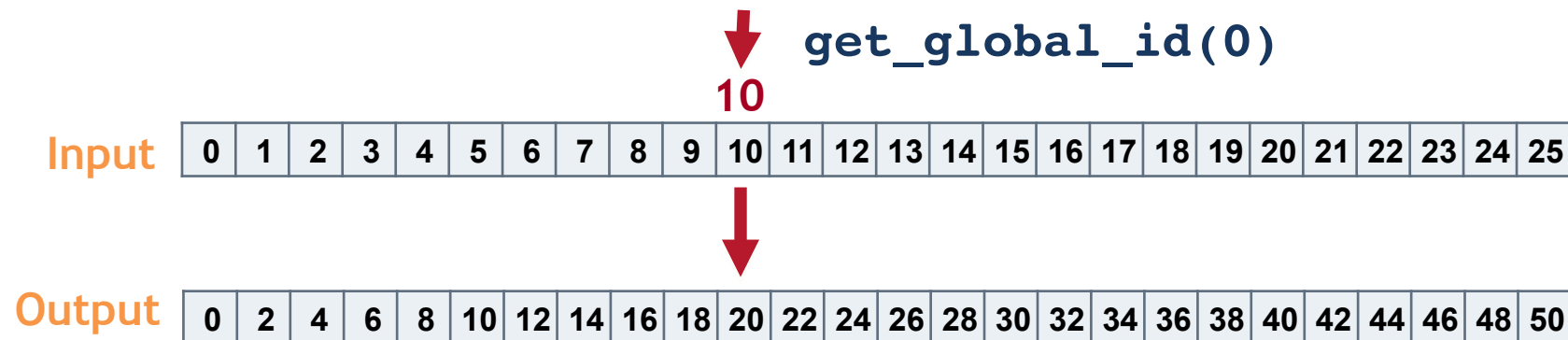
- **Context:**
 - The environment within which kernels execute and in which synchronization and memory management is defined.
- The **context** includes:
 - One or more devices
 - Device memory
 - One or more command-queues
- All **commands** for a device (kernel execution, synchronization, and memory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



Execution model (kernels)

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

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



Building Program Objects

- The program object encapsulates:
 - A context
 - The program source or binary, and
 - List of target devices and build options
- The build process to create a program object:
 - `clCreateProgramWithSource()`
 - `clCreateProgramWithBinary()`

OpenCL uses **runtime compilation** ... because in general you don't know the details of the target device when you ship the program

```
__kernel void
horizontal_reflect(read_only image2d_t src,
                  write_only image2d_t dst)
{
    int x = get_global_id(0); // x-coord
    int y = get_global_id(1); // y-coord
    int width = get_image_width(src);
    float4 src_val = read_imagef(src, sampler,
                                (int2)(width-1-x, y));
    write_imagef(dst, (int2)(x, y), src_val);
}
```

Compile for
GPU

GPU
code

Compile for
CPU

CPU
code

Example: vector addition

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

`C[i] = A[i] + B[i] for i=0 to N-1`

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

Vector Addition - Kernel

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

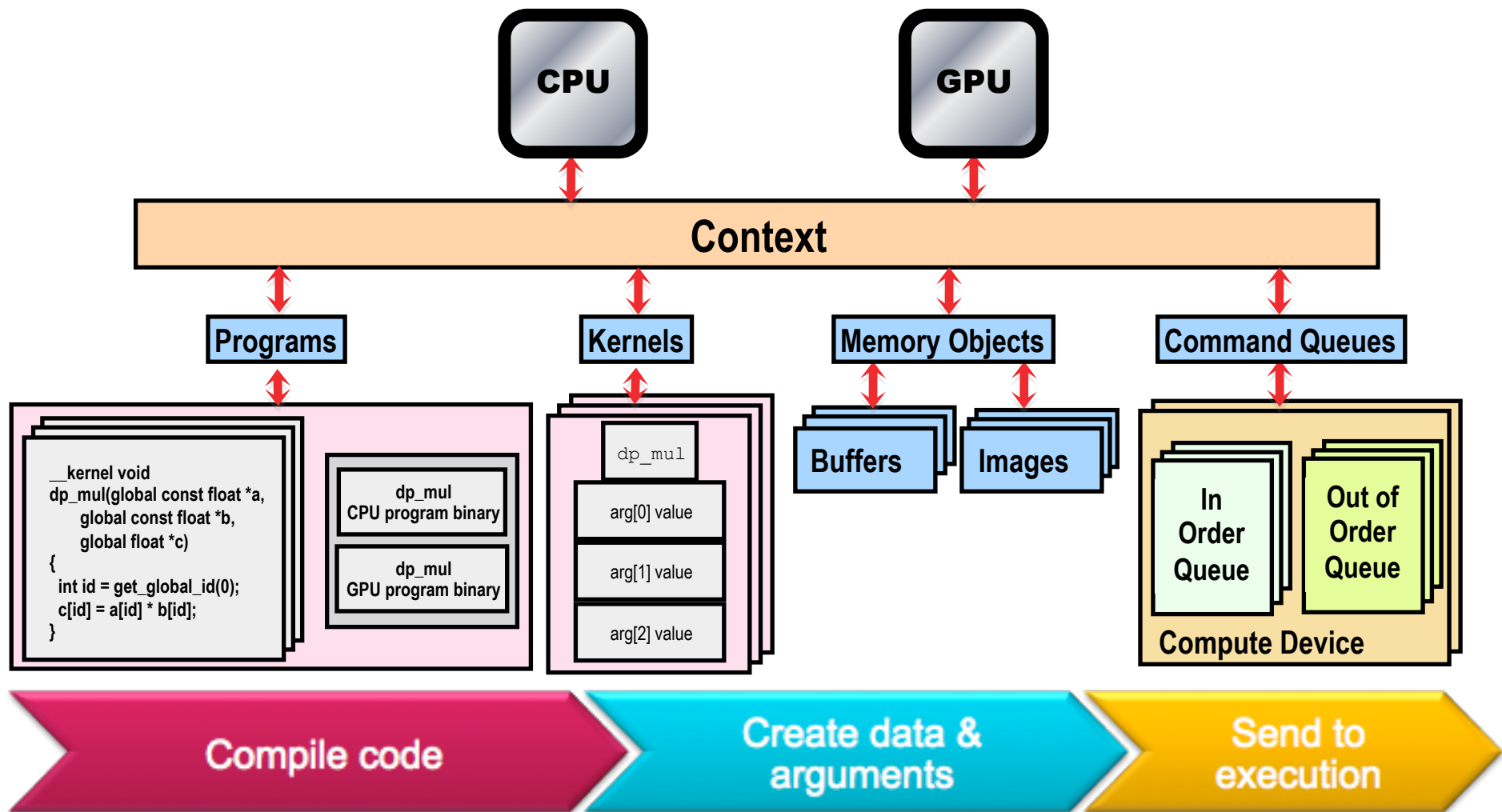
Exercise 1:

Running the Vector Add kernel

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

UNDERSTANDING THE HOST PROGRAM

The basic platform and runtime APIs in OpenCL



Vector Addition - Host

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



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

The C++ Interface

- Khronos has defined a common C++ header file containing a high level interface to OpenCL, [cl.hpp](#)
- This interface is dramatically easier to work with¹
- Key features:
 - Uses common defaults for the platform and command-queue, saving the programmer from extra coding for the most common use cases
 - Simplifies the basic API by bundling key parameters with the objects rather than requiring verbose and repetitive argument lists
 - Ability to “call” a kernel from the host, like a regular function
 - Error checking can be performed with C++ exceptions

¹ especially for C++ programmers...

C++ Interface: setting up the host program

- Enable OpenCL API **Exceptions**. Do this **before** including the header file

```
#define __CL_ENABLE_EXCEPTIONS
```

- Include key header files ... both standard and custom

```
#include <CL/cl.hpp>    // Khronos C++ Wrapper  
API
```

```
#include <cstdio>        // C style IO (e.g.  
printf)
```

```
#include <iostream>      // C++ style IO
```

```
#include <vector>         // C++ vector types  
For information about C++, see  
the appendix:
```

- Define key namespaces

```
using namespace cl;  
using namespace std.
```

“C++ for C programmers”.

1. Create a context and queue

- Grab a context using a device type:

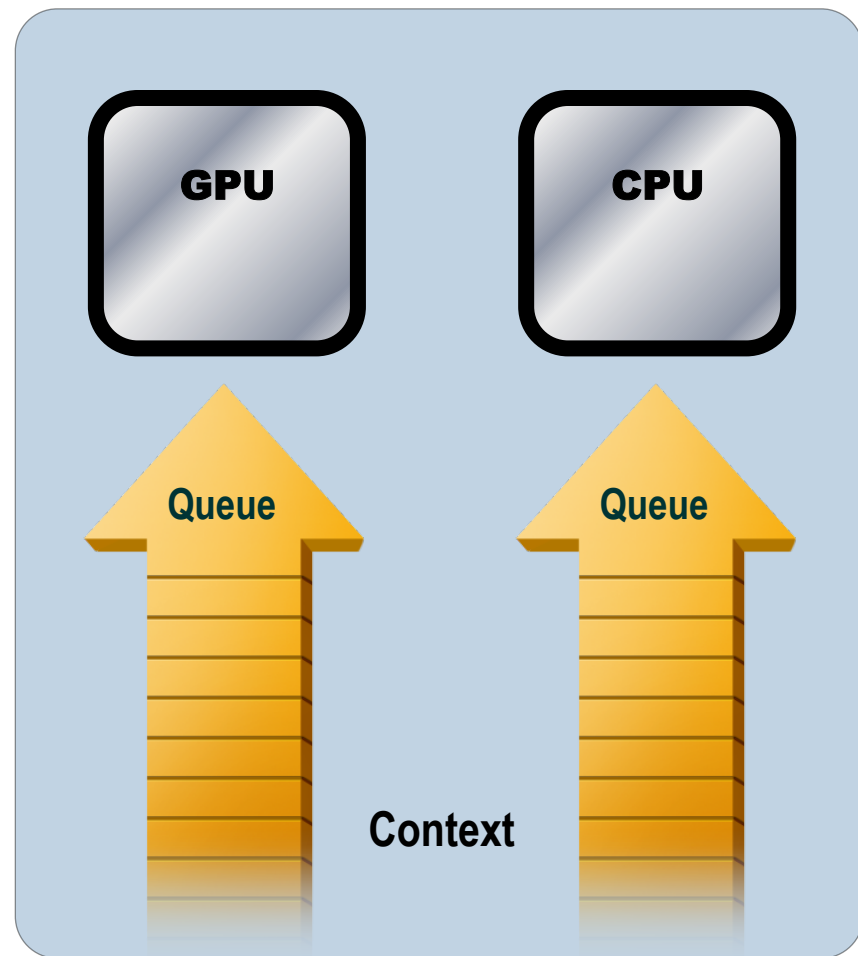
```
cl::Context context  
(CL_DEVICE_TYPE_DEFAULT);
```

- Create a command queue for the first device in the context:

```
cl::CommandQueue queue(context);
```

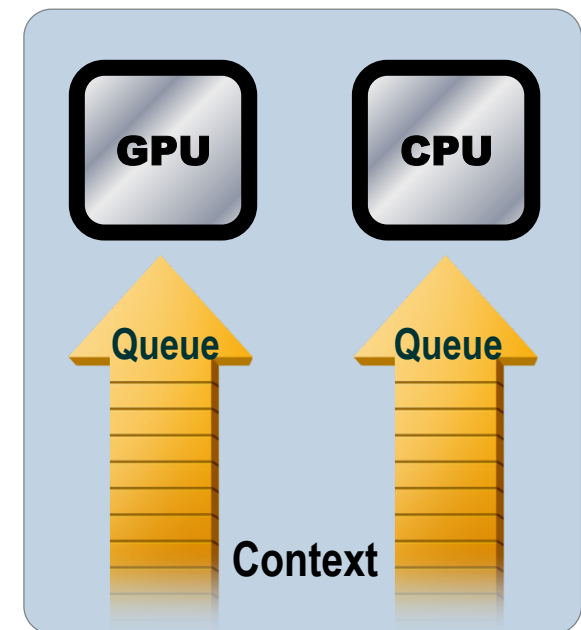
Command-Queues

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



Command-Queue execution details

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



2. Create and Build the program

- Define source code for the kernel-program either as a string literal (great for toy programs) or read it from a file (for real applications).
- Create the **program object and compile** to create a “dynamic library” from which specific kernels can be pulled:

```
cl::Program program(context, KernelSource, true);
```

3. Setup Memory Objects

- For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C

- Create input vectors and assign values **on the host**:

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

- Define **OpenCL** device buffers and copy from host buffers:

```
cl::Buffer d_a(context, begin(h_a), end(h_a), true);  
cl::Buffer d_b(context, begin(h_b), end(h_b), true);  
cl::Buffer d_c(context, CL_MEM_WRITE_ONLY,  
                    sizeof(float)*count);
```

What do we put in device memory?

- Memory Objects:
 - A handle to a reference-counted region of **global** memory.
- There are two kinds of memory object
 - **Buffer** object:
 - Defines a linear collection of bytes.
 - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
 - **Image** object:
 - Defines a two- or three-dimensional region of memory.
 - Image data can **only** be accessed with read and write functions, i.e. these are opaque data structures. The read functions use a sampler.

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

Creating and manipulating buffers

- Buffers are declared on the host as object type:

```
cl::Buffer
```

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

```
std::vector<float> h_a, h_b;
```

- Create the device-side **buffer** (d_a), assign read only memory to hold the host array (h_a) and copy it into device memory:

```
cl::Buffer
```

```
d_a(context, begin(h_a), end(h_a), true);
```

Creating and manipulating buffers

- Can specify device read/write access to the Buffer by setting the final argument to *false* instead of *true*
- Submit command to copy the device buffer back to host memory in array “h_c”:

```
cl::copy(queue, d_c, begin(h_c), end(h_c));
```

- Can also copy host memory to device buffers:

```
cl::copy(queue, begin(h_c), end(h_c), d_c);
```

4. Define the kernel

- Create a *kernel functor* for the kernels you want to be able to call in the *program*:

```
auto vadd =  
    cl::make_kernel  
        <cl::Buffer, cl::Buffer, cl::Buffer>  
        (program, "vadd");
```

- This means you can ‘call’ the kernel as a ‘function’ in your host code to enqueue the kernel.

5. Enqueue commands

- Specify *global* and *local* dimensions
 - `cl::NDRange global(1024)`
 - If you don't specify a local dimension, it is assumed as `cl::NullRange`, and the runtime picks a size for you
- Enqueue the kernel for execution (note: non-blocking):

```
vadd(cl::EnqueueArgs(queue, global), d_a, d_b, d_c);
```

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

```
cl::copy(queue, begin(h_c), end(h_c), d_c);
```

C++ interface: The vadd host program

```
#define N 1024
int main(void) {

vector<float> h_a(N), h_b(N), h_c(N);
// initialize these host vectors...

Buffer d_a, d_b, d_c;

Context
    context(CL_DEVICE_TYPE_DEFAULT);

CommandQueue queue(context);

Program
    program(
        context,
        loadprogram("vadd.cl"), true);

// Create the kernel functor
auto vadd = make_kernel
    <Buffer, Buffer, Buffer, int>
    (program, "vadd");

// Create buffers
// True indicates CL_MEM_READ_ONLY
// False indicates CL_MEM_READ_WRITE

d_a = Buffer(context, begin(h_a), end(h_a), true);
d_b = Buffer(context, begin(h_b), end(h_b), true);
d_c = Buffer(context, begin(h_c), end(h_c), false);

// Enqueue the kernel
vadd(EnqueueArgs(queue, NDRange(count)),
     d_a, d_b, d_c, count);

copy(queue, d_c, begin(h_c), end(h_c));

}
```

Note: The default context and command queue are used when we do not specify one in the function calls. The code here also uses the default device, so these cases are the same.

Exercise 2: Chaining vector add kernels

- **Goal:**
 - To verify that you understand manipulating kernel invocations and buffers in OpenCL
- **Procedure:**
 - Start with your VADD program in C++
 - Add additional buffer objects and assign them to vectors defined on the host (see the provided vadd programs for examples of how to do this)
 - Chain vadds ... e.g. $C=A+B$; $D=C+E$; $F=D+G$.
 - Read back the final result and verify that it is correct
- **Expected output:**
 - A message to standard output verifying that the chain of vector additions produced the correct result.

(Sample solution is for $C = A + B$; $D = C + E$; $F = D + G$; return F)

MODIFYING KERNELS

Working with Kernels (C++)


- The kernels are where all the action is in an OpenCL program.
- Steps to using kernels:
 1. Load kernel source code into a **program object** from a file
 2. Make a **kernel functor** from a function within the program
 3. Initialize **device memory**
 4. Call the **kernel functor**, specifying memory objects and global/local sizes
 5. Read **results** back from the device
- Note the kernel function argument list must match the kernel definition on the host.

Create a kernel

- Kernel code can be a string in the host code (toy codes)
- Or the kernel code can be loaded from a file (real codes)
- Compile for the default devices within the default context

```
program.build();
```

The build step can be carried out by specifying *true* in the program constructor. If you need to specify build flags you must specify *false* in the constructor and use this method instead.



- Define the kernel functor from a function within the program - allows us to 'call' the kernel to enqueue it

```
auto vadd = make_kernel<Buffer, Buffer, Buffer, int>  
            (program, "vadd");
```

- Advanced: if you want to query information about a kernel, you will need to create a kernel object:

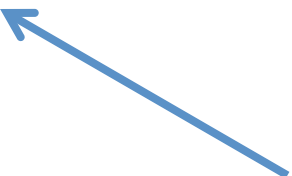
```
Kernel ko_vadd(program, "vadd");
```

If we set the local dimension ourselves or accept the OpenCL runtime's we don't need this step

Advanced: get info about the kernel

- E.g. get default size of local dimension (size of a Work-Group)

```
::size_t local =  
    ko_vadd.getWorkGroupInfo  
    <CL_KERNEL_WORK_GROUP_SIZE>  
    (Device::getDefault());
```

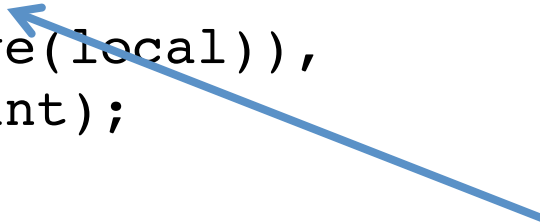


We can use any work-group-info parameter from table 5.15 in the OpenCL 1.1 specification. The function will return the appropriate type.

Call (enqueue) the kernel

- Enqueue the kernel for execution with buffer objects `d_a`, `d_b` and `d_c` and their length, `count`:

```
vadd(  
    EnqueueArgs(queue,  
                 NDRange(count),  
                 NDRange(local)),  
    d_a, d_b, d_c, count);
```

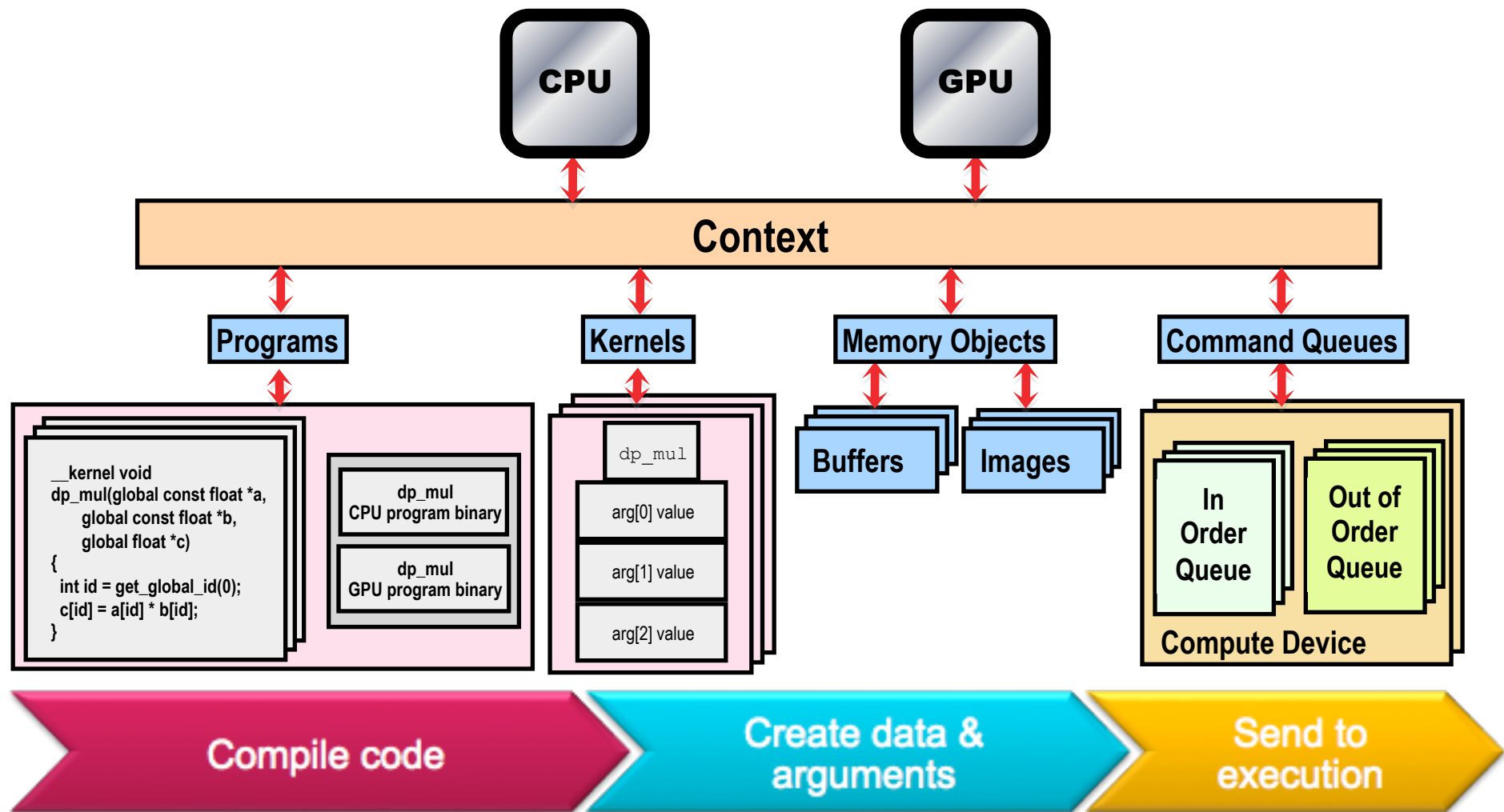


We can include any arguments from the `clEnqueueNDRangeKernel` function including Event wait lists (to be discussed later) and the command queue (optional)

Exercise 3: The $D = A + B + C$ problem

- **Goal:**
 - To verify that you understand how to control the argument definitions for a kernel.
 - To verify that you understand the host/kernel interface.
- **Procedure:**
 - Start with your VADD program.
 - Modify the kernel so it adds three vectors together.
 - Modify the host code to define three vectors and associate them with relevant kernel arguments.
 - Read back the final result and verify that it is correct.
- **Expected output:**
 - Test your result and verify that it is correct. Print a message to that effect on the screen.

We have now covered the basic platform runtime APIs in OpenCL



INTRODUCTION TO OPENCL KERNEL PROGRAMMING

OpenCL C kernel language

- Derived from **ISO C99**
 - A few *restrictions*: no recursion, function pointers, functions in C99 standard headers ...
 - Preprocessing directives defined by C99 are supported (#include etc.)
- Built-in data types
 - Scalar and vector data types, pointers
 - Data-type conversion functions:
 - `convert_type<_sat><_roundingmode>`
 - Image types: `image2d_t`, `image3d_t` and `sampler_t`

OpenCL C Language Highlights

- Function qualifiers
 - **__kernel** qualifier declares a function as a kernel
 - I.e. makes it visible to host code so it can be enqueued
 - Kernels can call other kernel-side functions
- Address space qualifiers
 - **__global**, **__local**, **__constant**, **__private**
 - Pointer kernel arguments must be declared with an address space qualifier
- Work-item functions
 - **get_work_dim()**, **get_global_id()**, **get_local_id()**, **get_group_id()**
- Synchronization functions
 - **Barriers** - all work-items within a work-group must execute the barrier function before any work-item can continue
 - **Memory fences** - provides ordering between memory operations

OpenCL C Language Restrictions

- Pointers to functions are *not* allowed
- Pointers to pointers allowed *within* a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are *optional* in OpenCL v1.1, but the key word is reserved
(note: most implementations support double)

Matrix multiplication: sequential code

We calculate $C=AB$, $\text{dimA} = (N \times P)$, $\text{dimB}=(P \times M)$, $\text{dimC}=(N \times M)$

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

$$C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j]$$

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

```
}
```

Matrix multiplication performance

- Serial C code on CPU (single core).

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

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

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

Matrix multiplication: sequential code

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

We turn this into an OpenCL kernel!

Matrix multiplication: OpenCL kernel (1/2)

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

Mark as a kernel function and
specify memory qualifiers

Matrix multiplication: OpenCL kernel (2/2)

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

Remove outer loops and set
work-item co-ordinates

Matrix multiplication: OpenCL kernel improved

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

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

Matrix multiplication host program (C++ API)

```
int main(int argc, char *argv[])
{
    std::vector<float> h_A, h_B, h_C; // matrices
    int Mdim, Ndim, Pdim; // A[N][P], B[P][M], C[N][M]
    int i, err;
    int szA, szB, szC; // num elements in each matrix
    double start_time, run_time; // timing data
    cl::Program program;

    Ndim = Pdim = Mdim = ORDER;
    szA = Ndim*Pdim;
    szB = Pdim*Mdim;
    szC = Ndim*Mdim;
    h_A = std::vector<float>(szA);
    h_B = std::vector<float>(szB);
    h_C = std::vector<float>(szC);

    initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);

    // Compile for first kernel to setup program
    program = cl::Program(C_elem_KernelSource, true);
    Context context(CL_DEVICE_TYPE_DEFAULT);
    cl::CommandQueue queue(context);
    std::vector<Device> devices =
        context.getInfo<CL_CONTEXT_DEVICES>();
    cl::Device device = devices[0];
    std::string s =
        device.getInfo<CL_DEVICE_NAME>();
    std::cout << "\nUsing OpenCL Device: " << s << "\n";

    // Setup the buffers, initialize matrices,
    // and write them into global memory
    initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);
    cl::Buffer d_a(context, begin(h_A), end(h_A), true);
    cl::Buffer d_b(context, begin(h_B), end(h_B), true);
    cl::Buffer d_c = cl::Buffer(context,
                                CL_MEM_WRITE_ONLY,
                                sizeof(float) * szC);

    auto naive = cl::make_kernel<int, int, int,
                                cl::Buffer, cl::Buffer, cl::Buffer>
        (program, "mmul");

    zero_mat(Ndim, Mdim, h_C);
    start_time = wtime();

    naive(cl::EnqueueArgs(queue,
                           cl::NDRange(Ndim, Mdim)),
          Ndim, Mdim, Pdim, d_a, d_b, d_c);

    cl::copy(queue, d_c, begin(h_C), end(h_C));

    run_time = wtime() - start_time;
    results(Mdim, Ndim, Pdim, h_C, run_time);
}
```

Note: To use the default context/queue/device, skip this section and remove the references to context, queue and device.

Matrix multiplication host program (C++ API)

```
int main(int argc, char *argv[])
{
    std::vector<float> h_A, h_B, h_C; // matrices
    int Mdim, Ndim, Pdim; // A[N][P], B[P][M], C[N][M]
    int i, err;
    int szA, szB, szC; // num elements in each matrix
    double start_time, run_time; // timing data
    cl::Program program;

    Ndim = Pdim = Mdim = ORDER;
    szA = Ndim*Pdim;
    szB = Pdim*Mdim;
    szC = Ndim*Mdim;
    h_A = std::vector<float>(szA);
    h_B = std::vector<float>(szB);
    h_C = std::vector<float>(szC);

    initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);

    // Compile for first kernel to setup program
    program = cl::Program::create(context, source, true);
    Context context;
    cl::CommandQueue queue(context, device, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);
    std::vector<cl::Buffer> buffers;
    cl::Device device;
    std::string s =
        device.getInfo<CL_DEVICE_NAME>();
    std::cout << "\nUsing OpenCL Device " << device.getInfo<CL_DEVICE_NAME>() << "\n";
}
```

Declare and initialize data

Setup the platform and build program

Setup buffers and write A and B matrices to the device memory

```
// Setup the
// and write
initmat(Mdim
cl::Buffer d
cl::Buffer d_b(context, begin(h_B), end(h_B), true);
cl::Buffer d_c = cl::Buffer(context,
                        CL_MEM_WRITE_ONLY,
                        sizeof(float) * szC);
```

```
auto naive = cl::make_kernel<int, int, int,
                        cl::Buffer, cl::Buffer, cl::Buffer>
(program, "mmul");
```

Create the kernel functor

```
zero_r
start_time = wtime();
```

Run the kernel and collect results

```
naive(cl::EnqueueArgs(queue,
                        Ndims, Mdim, Pdim),
      h_A, h_B, h_C);
cl::copy(queue, h_C, results);
```

```
run_time = wtime() - start_time;
results(Mdim, Ndim, Pdim, h_C, run_time);
```

Note: To use the default context/queue/device, skip this section and remove the references to context, queue and device.

Matrix multiplication performance

- Matrices are stored in global memory.

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

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

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

Third party names are the property of their owners.

UNDERSTANDING THE OPENCL MEMORY HIERARCHY

Optimizing matrix multiplication

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

The diagram illustrates the dot-product algorithm for matrix multiplication. It shows the calculation of a single element $C(i,j)$ as the dot product of a row of matrix A and a column of matrix B. The equation is represented as:

$$\boxed{\begin{matrix} & & C(i,j) \\ & \blacksquare & \end{matrix}} = \boxed{\begin{matrix} & & C(i,j) \\ & \blacksquare & \end{matrix}} + \boxed{\begin{matrix} A(i,:) \\ \hline \end{matrix}} \times \boxed{\begin{matrix} \hline B(:,j) \end{matrix}}$$

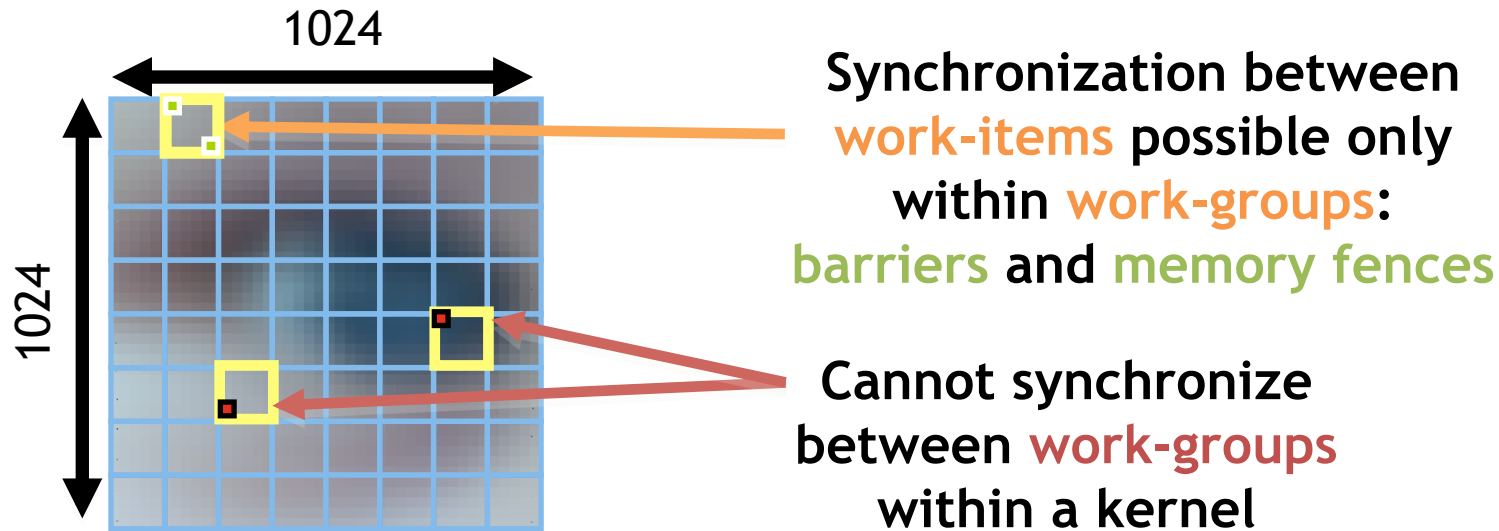
In the diagram, the red square in the first box represents the element $C(i,j)$. The second box is identical. The third box shows a row of matrix A, $A(i,:)$, highlighted with a red horizontal line. The fourth box shows a column of matrix B, $B(:,j)$, highlighted with a red vertical line. The multiplication is indicated by a bold 'x'.

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

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

An N-dimensional domain of work-items

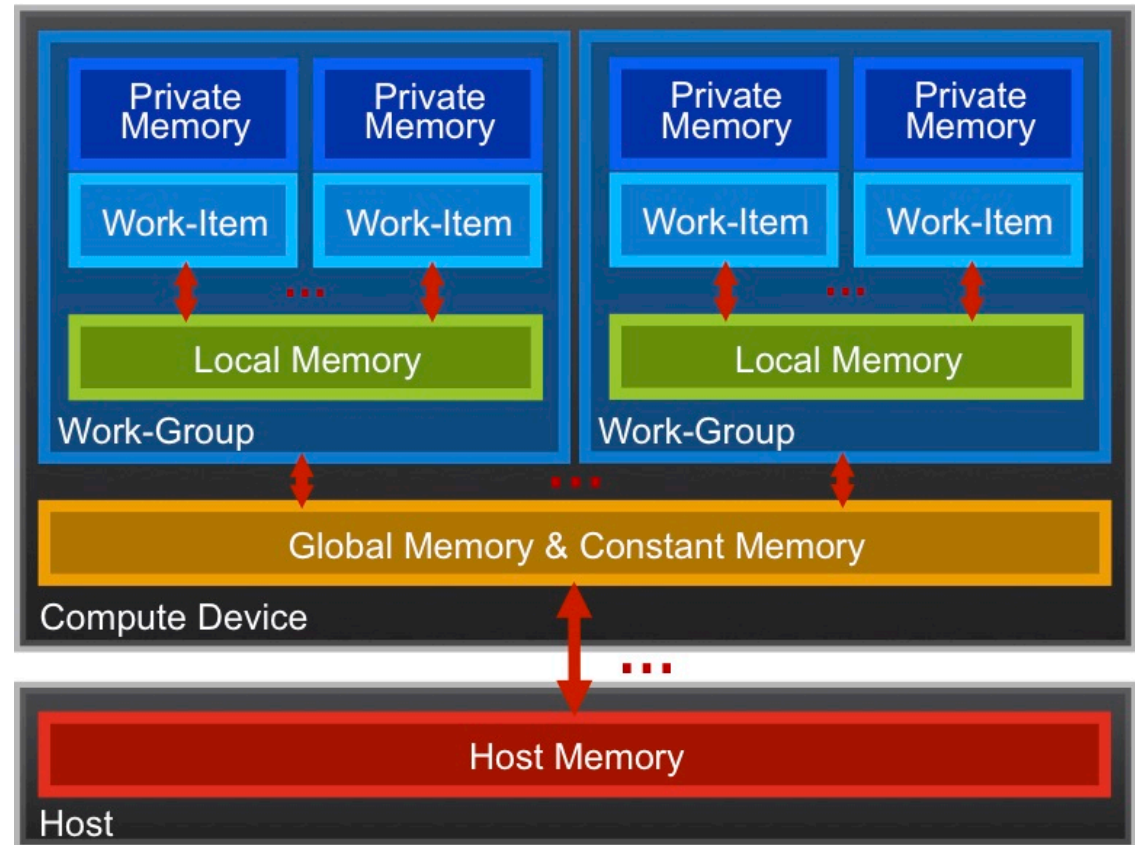
- **Global** Dimensions:
 - 1024x1024 (whole problem space)
- **Local** Dimensions:
 - 128x128 (**work-group**, executes together)



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

OpenCL Memory model

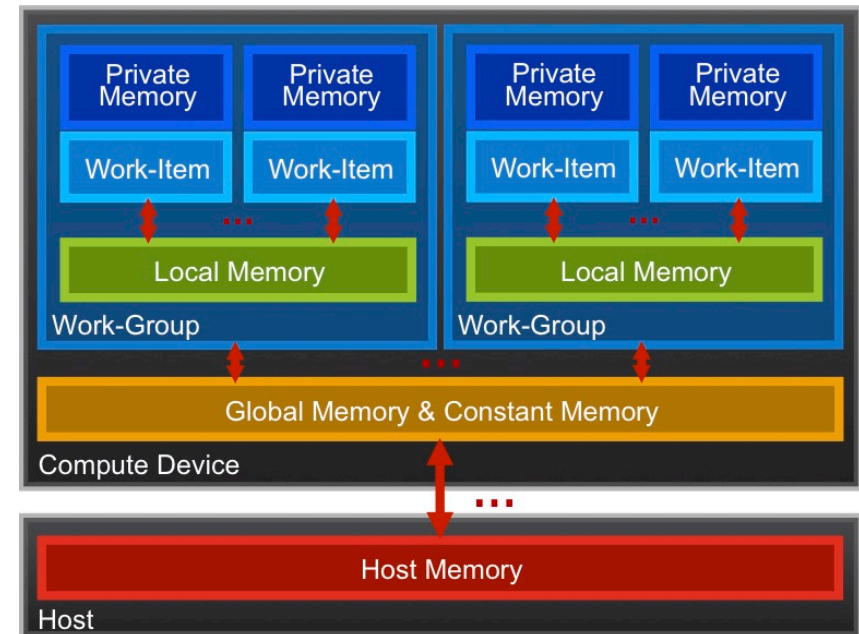
- **Private Memory**
 - Per work-item
- **Local Memory**
 - Shared within a work-group
- **Global/Constant Memory**
 - Visible to all work-groups
- **Host memory**
 - On the CPU



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

OpenCL Memory model

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



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

Private Memory

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

* Occupancy on a GPU

Local Memory

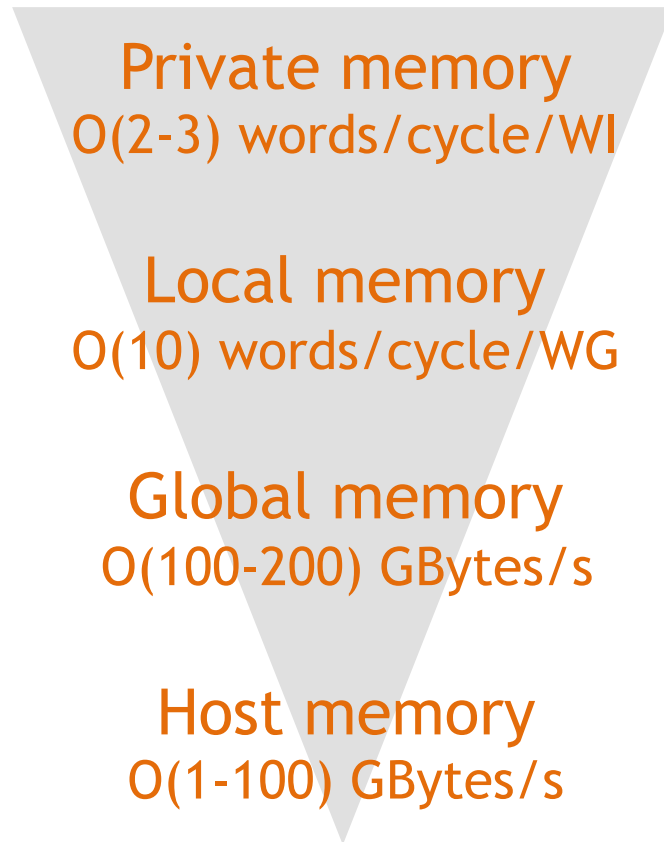
- Tens of KBytes per Compute Unit
 - As multiple Work-Groups will be running on each CU, this means only a fraction of the total Local Memory size is available to each Work-Group
- Assume $O(1-10)$ KBytes of Local Memory per Work-Group
 - Your kernels are responsible for transferring data between Local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

Local Memory

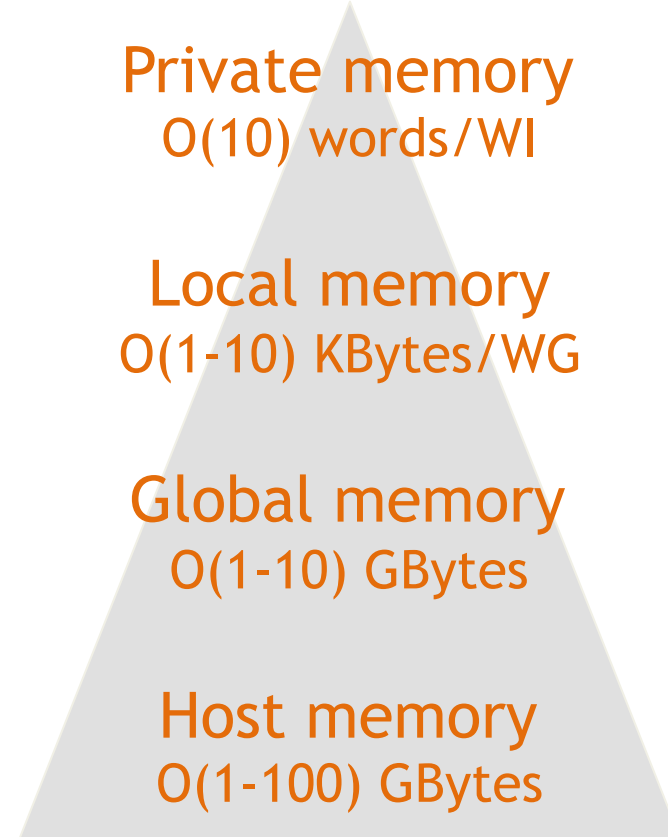
- **Local Memory** doesn't always help...
 - CPUs don't have special hardware for it
 - This can mean excessive use of Local Memory might slow down kernels on CPUs
 - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
 - So,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

The Memory Hierarchy

Bandwidths



Sizes



Speeds and feeds approx. for a high-end discrete GPU, circa 2011

Memory Consistency

- OpenCL uses a **relaxed consistency** memory model; i.e.
 - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
 - Memory has load/store consistency to the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group:
 - Local memory is consistent between work-items at a barrier.
- Global memory is consistent within a work-group at a barrier, but **not** guaranteed across different work-groups!!
 - This is a common source of bugs!
- Consistency of memory shared between **commands** (e.g. kernel invocations) is enforced by **synchronization** (barriers, events, in-order queue)

Optimizing matrix multiplication

- There may be significant overhead to manage work-items and work-groups.
- So let's have each work-item compute a full row of C

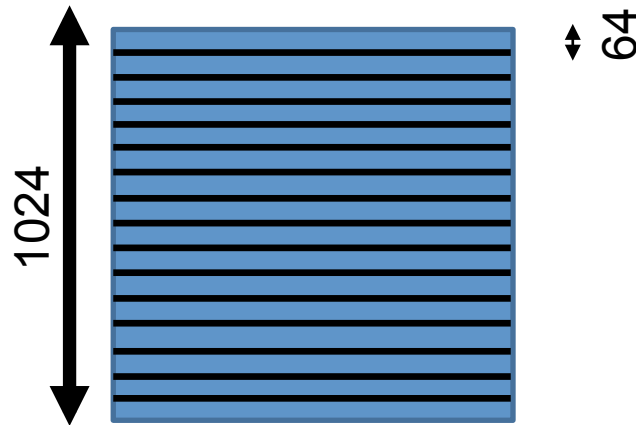
The diagram illustrates the calculation of a single element $C(i,j)$ in matrix multiplication. It shows the following components:

- A square matrix representing C with a red horizontal band across its middle. A small brown square is located within this band, labeled $C(i,j)$.
- An equals sign ($=$) followed by another identical square matrix representing C with the same red band and brown square labeled $C(i,j)$.
- A plus sign ($+$) followed by a square matrix representing A with a red horizontal band across its middle, labeled $A(i,:)$.
- A multiplication sign (\times) followed by a square matrix representing B with a red vertical band down its middle, labeled $B(:,j)$.

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

An N-dimension domain of work-items

- **Global** Dimensions: 1024 (1D)
Whole problem space (index space)
- **Local** Dimensions: 64 (work-items per work-group)
Only $1024/64 = 16$ work-groups in total



- Important implication: we will have a lot fewer work-items per work-group (64) and work-groups (16). Why might this matter?

Reduce work-item overhead

Do a whole row of C per work-item

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

Matrix multiplication host program (C++ API)

Changes to host program:

1. 1D ND Range set to number of rows in the C matrix
2. Local Dimension set to 64 so number of work-groups match number of compute units (16 in this case) for our order 1024 matrices

```
int main(
{
    std::vector<float> h_A, h_B, h_C;
    int Mdim = 1024;
    int i, j, k;
    int szA = Mdim * Pdim;
    double szB = Mdim * Pdim;
    cl::Program program;

    Ndim = Pdim = Mdim = ORDER;
    szA = Ndim * Pdim;
    szB = Pdim * Mdim;
    szC = Ndim * Mdim;
    h_A = std::vector<float>(szA);
    h_B = std::vector<float>(szB);
    h_C = std::vector<float>(szC);

    initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);

    // Compile for first kernel to setup program
    program = cl::Program(C_elem_KernelSource, true);
    Context context(CL_DEVICE_TYPE_DEFAULT);
    cl::CommandQueue queue(context);
    std::vector<Device> devices =
        context.getInfo<CL_CONTEXT_DEVICES>();
    cl::Device device = devices[0];
    std::string s =
        device.getInfo<CL_DEVICE_NAME>();
    std::cout << "\nUsing OpenCL Device "
        << s << "\n";

    auto krow = cl::make_kernel<int, int, int,
        cl::Buffer, cl::Buffer, cl::Buffer>
        (program, "mmul");

    zero_mat(Ndim, Mdim, h_C);
    start_time = wtime();

    krow(cl::EnqueueArgs(queue
        cl::NDRange(Ndim),
        cl::NDRange(ORDER/16)),
        Ndim, Mdim, Pdim, a_in, b_in, c_out);

    cl::copy(queue, d_c, begin(h_C), end(h_C));

    run_time = wtime() - start_time;
    results(Mdim, Ndim, Pdim, h_C, run_time);
}
```

Matrix multiplication host program (C++ API)

```
int main(int argc, char *argv[])
{
    std::vector<float> h_A, h_B, h_C; // matrices
    int Mdim, Ndim, Pdim; // A[N][P], B[P][M], C[N][M]
    int i, err;
    int szA, szB, szC; // num elements in each matrix
    double start_time, run_time; // timing data
    cl::Program program;

    Ndim = Pdim = Mdim = ORDER;
    szA = Ndim*Pdim;
    szB = Pdim*Mdim;
    szC = Ndim*Mdim;
    h_A = std::vector<float>(szA);
    h_B = std::vector<float>(szB);
    h_C = std::vector<float>(szC);

    initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);

    // Compile for first kernel to setup program
    program = cl::Program(C_elem_KernelSource, true);
    Context context(CL_DEVICE_TYPE_DEFAULT);
    cl::CommandQueue queue(context);
    std::vector<Device> devices =
        context.getInfo<CL_CONTEXT_DEVICES>();
    cl::Device device = devices[0];
    std::string s =
        device.getInfo<CL_DEVICE_NAME>();
    std::cout << "\nUsing OpenCL Device "
        << s << "\n";

    // Setup the buffers, initialize matrices,
    // and write them into global memory
    initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);
    cl::Buffer d_a(context, begin(h_A), end(h_A), true);
    cl::Buffer d_b(context, begin(h_B), end(h_B), true);
    cl::Buffer d_c = cl::Buffer(context,
                                CL_MEM_WRITE_ONLY,
                                sizeof(float) * szC);

    auto krow = cl::make_kernel<int, int, int,
                                cl::Buffer, cl::Buffer, cl::Buffer>
        (program, "mmul");

    zero_mat(Ndim, Mdim, h_C);
    start_time = wtime();

    krow(cl::EnqueueArgs(queue,
                          cl::NDRange(Ndim),
                          cl::NDRange(ORDER/16)),
        Ndim, Mdim, Pdim, a_in, b_in, c_out);

    cl::copy(queue, d_c, begin(h_C), end(h_C));

    run_time = wtime() - start_time;
    results(Mdim, Ndim, Pdim, h_C, run_time);
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8

This has started to help. 

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

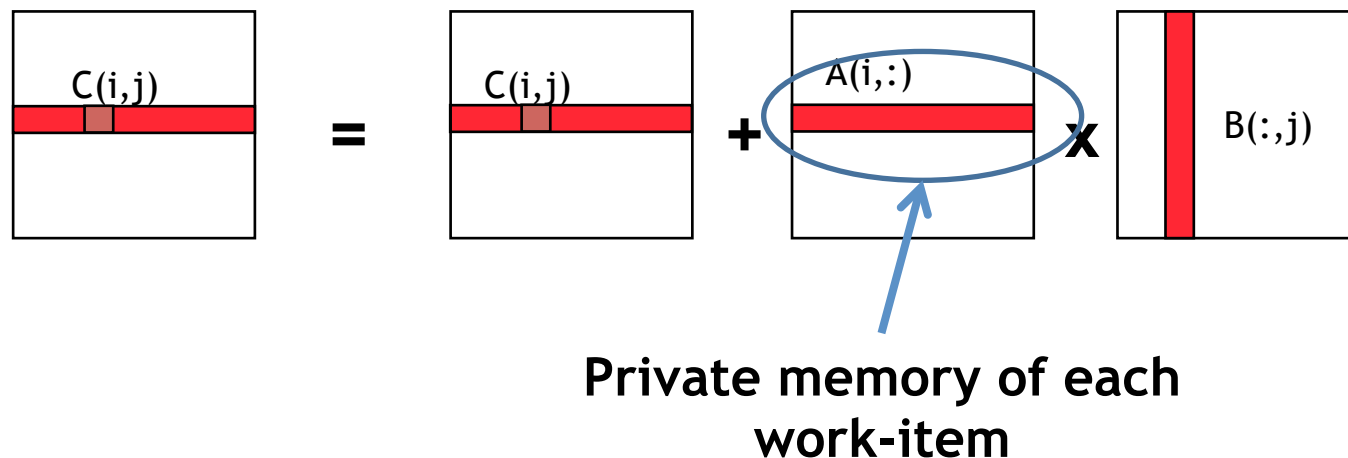
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

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

Third party names are the property of their owners.

Optimizing matrix multiplication

- Notice that, in one row of C , each element reuses the same row of A .
- Let's copy that row of A into private memory of the work-item that's (exclusively) using it to avoid the overhead of loading it from global memory for each $C(i,j)$ computation.



Matrix multiplication: OpenCL kernel (3/3)

```
__kernel void mmul(  
    const int Mdim,  
    const int Ndim,  
    const int Pdim,  
    __global float *A,  
    __global float *B,  
    __global float *C)  
{  
    int k, j;  
    int i = get_global_id(0);  
    float Awrk[1024];  
    float tmp;
```

```
        for (k = 0; k < Pdim; k++)  
            Awrk[k] = A[i*Ndim+k];  
        for (j = 0; j < Mdim; j++) {  
            tmp = 0.0f;  
            for (k = 0; k < Pdim; k++)  
                tmp += Awrk[k]*B[k*Pdim+j];  
            C[i*Ndim+j] += tmp;  
        }  
    }
```

Setup a work array for A in private memory and copy into it from global memory before we start with the matrix multiplications.

(Actually, this is using *far* more private memory than we'll have and so Awrk[] will be spilled to global memory)

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3

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

Big impact!

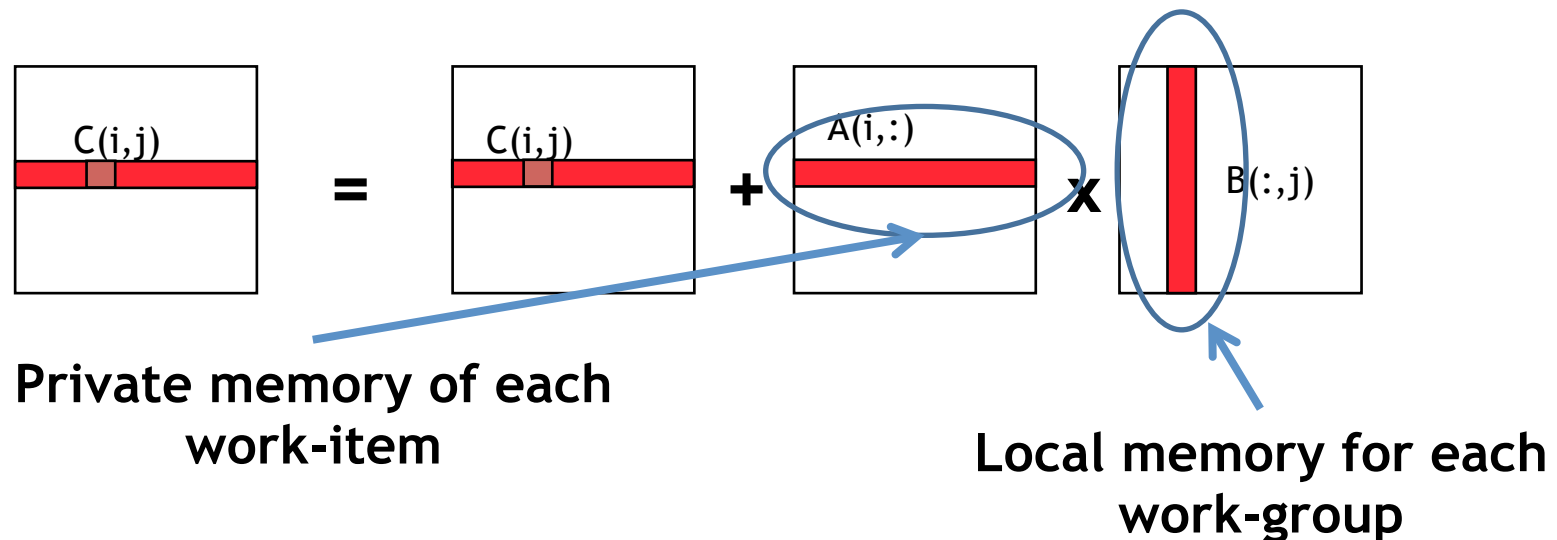


These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Third party names are the property of their owners.

Optimizing matrix multiplication

- We already noticed that, in one row of C , each element uses the same row of A
- Each work-item in a work-group also uses the same columns of B
- So let's store the B columns in **local** memory (which is shared by the work-items in the work-group)



Row of C per work-item, A row private, B columns local

```
__kernel void mmul(  
    const int Mdim,  
    const int Ndim,  
    const int Pdim,  
    __global float *A,  
    __global float *B,  
    __global float *C,  
    __local float *Bwrk)  
{  
    int k, j;  
    int i = get_global_id(0);  
    int iloc = get_local_id(0);  
    int nloc = get_local_size(0);  
    float Awrk[1024];
```

```
    float tmp;  
    for (k = 0; k < Pdim; k++)  
        Awrk[k] = A[i*Ndim+k];  
    for (j = 0; j < Mdim; j++) {  
        for (k=iloc; k<Pdim; k+=nloc)  
            Bwrk[k] = B[k*Pdim+j];  
        barrier(CLK_LOCAL_MEM_FENCE);  
        tmp = 0.0f;  
        for (k = 0; k < Pdim; k++)  
            tmp += Awrk[k] * Bwrk[k];  
        C[i*Ndim+j] += tmp;  
    }  
}
```

Pass in a pointer to local memory. Work-items in a work-group start by copying the columns of B they need into their local memory.

Matrix multiplication host program (C++ API)

Changes to host program:

1. Pass local memory to kernels.
 1. This requires a change to the kernel argument lists ... an arg of type LocalSpaceArg is needed.
2. Allocate the size of local memory
3. Update argument list in kernel functor

```
int main
{
    std::vector<float> h_A, h_B, h_C;
    int Mdim, Ndim, Pdim;
    int i, j, k;
    int szA, szB, szC;
    double t;
    cl::Program program;

    Ndim = Pdim = Mdim = ORDER;
    szA = Ndim*Pdim;
    szB = Pdim*Mdim;
    szC = Ndim*Mdim;
    h_A = std::vector<float>(szA);
    h_B = std::vector<float>(szB);
    h_C = std::vector<float>(szC);

    initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);

    // Compile for first kernel to setup program
    program = cl::Program(C_elem_KernelSource, true);
    Context context(CL_DEVICE_TYPE_DEFAULT);
    cl::CommandQueue queue(context);
    std::vector<Device> devices =
        context.getInfo<CL_CONTEXT_DEVICES>();
    cl::Device device = devices[0];
    std::string s =
        device.getInfo<CL_DEVICE_NAME>();
    std::cout << "\nUsing OpenCL Device "
        << s << "\n";
}
```

```
cl::LocalSpaceArg localmem =
    cl::Local(sizeof(float) * Pdim);
```

```
auto rowcol = cl::make_kernel<int, int, int,
    cl::Buffer, cl::Buffer, cl::Buffer,
    cl::LocalSpaceArg>(program, "mmul");
```

```
zero_mat(Ndim, Mdim, h_C);
start_time = wtime();
```

```
rowcol(cl::EnqueueArgs(queue,
    cl::NDRange(Ndim),
    cl::NDRange(ORDER/16)),
    Ndim, Mdim, Pdim, d_a, d_b, d_c, localmem);
```

```
cl::copy(queue, d_c, begin(h_C), end(h_C));
```

```
run_time = wtime() - start_time;
results(Mdim, Ndim, Pdim, h_C, run_time);
```

```
}
```

Matrix multiplication host program (C++ API)

```
int main(int argc, char *argv[])
{
    std::vector<float> h_A, h_B, h_C; // matrices
    int Mdim, Ndim, Pdim; // A[N][P], B[P][M], C[N][M]
    int i, err;
    int szA, szB, szC; // num elements in each matrix
    double start_time, run_time; // timing data
    cl::Program program;

    Ndim = Pdim = Mdim = ORDER;
    szA = Ndim*Pdim;
    szB = Pdim*Mdim;
    szC = Ndim*Mdim;
    h_A = std::vector<float>(szA);
    h_B = std::vector<float>(szB);
    h_C = std::vector<float>(szC);

    initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);

    // Compile for first kernel to setup program
    program = cl::Program(C_elem_KernelSource, true);
    Context context(CL_DEVICE_TYPE_DEFAULT);
    cl::CommandQueue queue(context);
    std::vector<Device> devices =
        context.getInfo<CL_CONTEXT_DEVICES>();
    cl::Device device = devices[0];
    std::string s =
        device.getInfo<CL_DEVICE_NAME>();
    std::cout << "\nUsing OpenCL Device "
        << s << "\n";
```

```
// Setup the buffers, initialize matrices,
// and write them into global memory
initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);
cl::Buffer d_a(context, begin(h_A), end(h_A), true);
cl::Buffer d_b(context, begin(h_B), end(h_B), true);
cl::Buffer d_c = cl::Buffer(context,
                             CL_MEM_WRITE_ONLY,
                             sizeof(float) * szC);
```

```
cl::LocalSpaceArg localmem =
    cl::Local(sizeof(float) * Pdim);
```

```
auto rowcol = cl::make_kernel<int, int, int,
                             cl::Buffer, cl::Buffer, cl::Buffer,
                             cl::LocalSpaceArg>(program, "mmul");
```

```
zero_mat(Ndim, Mdim, h_C);
start_time = wtime();
```

```
rowcol(cl::EnqueueArgs(queue,
                        cl::NDRange(Ndim),
                        cl::NDRange(ORDER/16)),
        Ndim, Mdim, Pdim, d_a, d_b, d_c, localmem);
```

```
cl::copy(queue, d_c, begin(h_C), end(h_C));
```

```
run_time = wtime() - start_time;
results(Mdim, Ndim, Pdim, h_C, run_time);
```

```
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3
C row per work-item, A private, B local	10,047.5	8,181.9

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

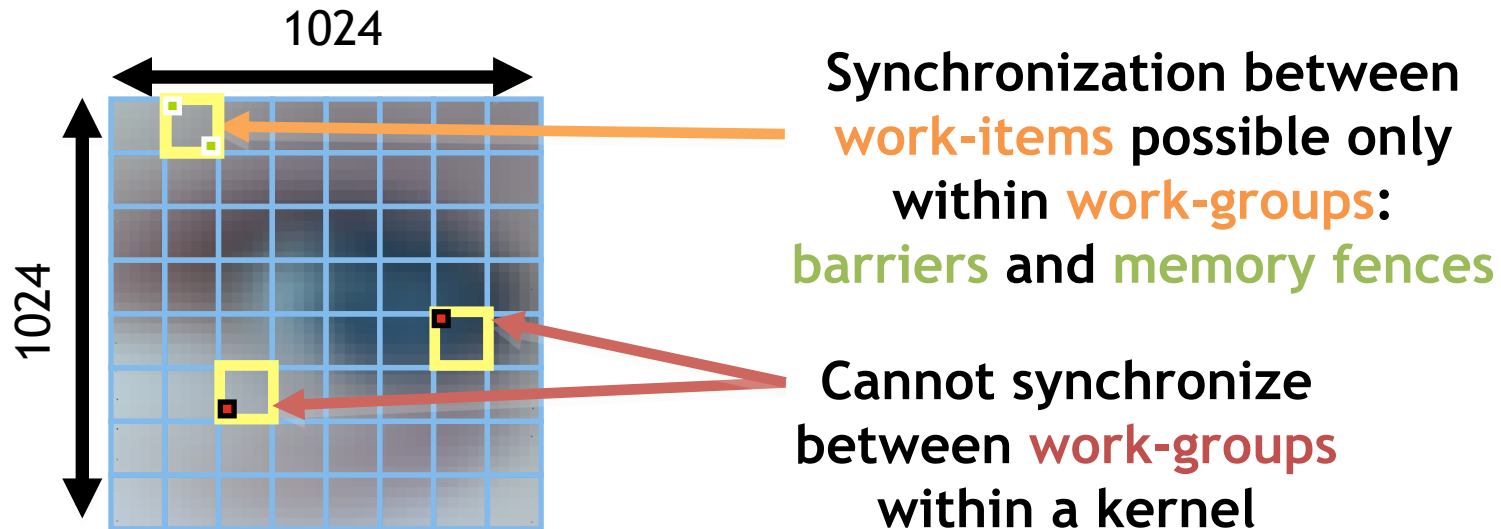
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Third party names are the property of their owners.

SYNCHRONIZATION IN OPENCL

Consider N-dimensional domain of work-items

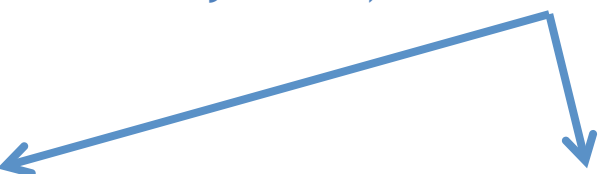
- **Global Dimensions:**
 - 1024x1024 (whole problem space)
- **Local Dimensions:**
 - 128x128 (**work-group**, executes together)



Synchronization: when multiple units of execution (e.g. work-items) are brought to a known point in their execution. Most common example is a barrier ... i.e. all units of execution “in scope” arrive at the **barrier** before any proceed.

Work-Item Synchronization

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)



- Within a work-group

void barrier()

- Takes optional flags

CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE

- A work-item that encounters a **barrier()** will wait until ALL work-items in its work-group reach the **barrier()**
- **Corollary:** If a **barrier()** is inside a branch, then the branch **must** be taken by either:
 - **ALL** work-items in the work-group, OR
 - **NO** work-item in the work-group

- Across work-groups

- No guarantees as to where and when a particular work-group will be executed relative to another work-group
- Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)
- **Only solution: finish the kernel and start another**

Where might we need synchronization?

- Consider a reduction ... reduce a set of numbers to a single value
 - E.g. find sum of all elements in an array
- Sequential code

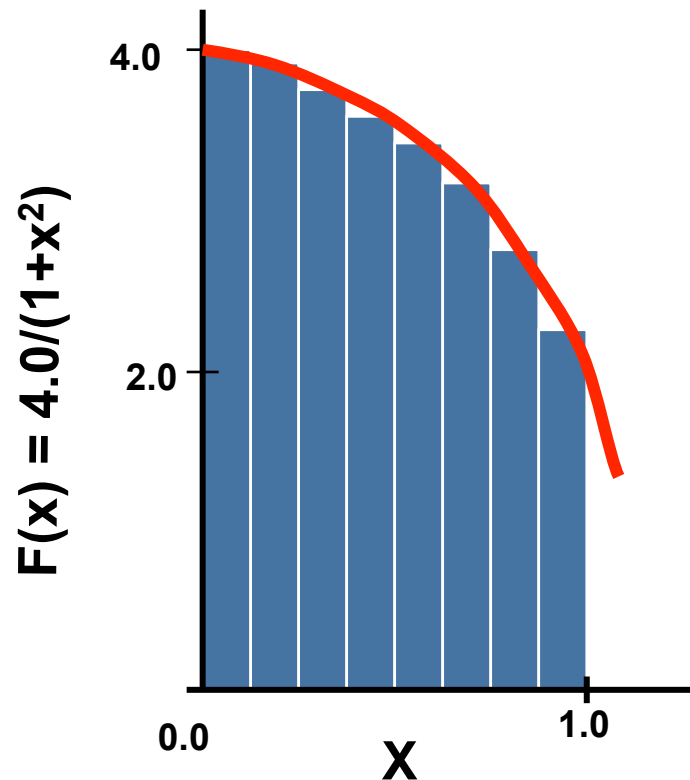
```
int reduce(int Ndim, int *A)
{
    sum = 0;
    for(int i = 0; i < Ndim; i++)
        sum += A[i];
}
```

Simple parallel reduction

- A reduction can be carried out in three steps:
 1. Each work-item sums its private values into a local array indexed by the work-item's local id
 2. When all the work-items have finished, one work-item sums the local array into an element of a global array (indexed by work-group id).
 3. When all work-groups have finished the kernel execution, the global array is summed on the host.
- Note: this is a simple reduction that is straightforward to implement. More efficient reductions do the work-group sums in parallel on the device rather than on the host. These more scalable reductions are considerably more complicated to implement.

A simple program that uses a reduction

Numerical Integration



Mathematically, we know that we can approximate the integral as a sum of rectangles.

Each rectangle has width and height at the middle of interval.

Numerical integration source code

The serial Pi program

```
static long num_steps = 100000;
double step;
void main()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i = 0; i < num_steps; i++) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Exercise 4: The Pi program

- **Goal:**
 - To understand synchronization between work-items in the OpenCL C kernel programming language
- **Procedure:**
 - Start with the provided serial program to estimate Pi through numerical integration
 - Write a kernel and host program to compute the numerical integral using OpenCL
 - Note: You will need to implement a reduction
- **Expected output:**
 - Output result plus an estimate of the error in the result
 - Report the runtime

Hint: you will want each work-item to do many iterations of the loop, i.e. don't create one work-item per loop iteration. To do so would make the reduction so costly that performance would be terrible.

SOME CONCLUDING REMARKS

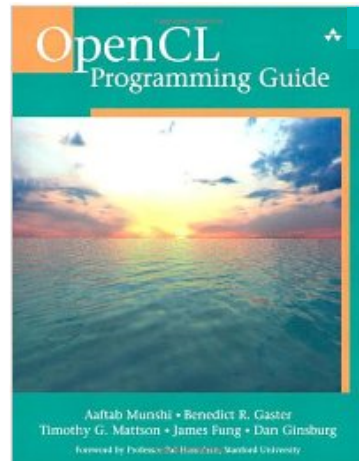
Conclusion

- OpenCL has *widespread* industrial support
- OpenCL defines a platform-API/framework for *heterogeneous computing*, not just GPGPU or CPU-offload programming
- OpenCL has the potential to deliver *portably performant code*; but it has to be used correctly
- The latest *C++ and Python APIs* makes developing OpenCL programs much simpler than before
- The future is clear:
 - OpenCL is the *only* parallel programming standard that enables mixing task parallel and data parallel code in a single program while load balancing across **ALL** of the platform's available resources.

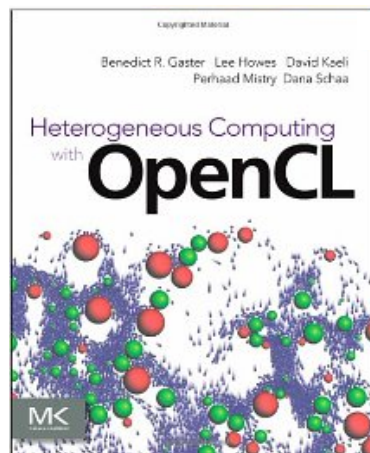
Other important related trends

- The Heterogeneous Systems Architecture, HSA
 - New standard supported by HSA Foundation (hsafoundation.com)
 - Partners include Samsung, ARM, IMG, Qualcomm, LG, TI, Mediatek, ...
- OpenCL's Standard Portable Intermediate Representation (SPIR)
 - Based on LLVM's IR
 - Makes interchangeable front- and back-ends straightforward
- OpenCL 2.0
 - Adding High Level Model (HLM)
 - Lots of other improvements
- For the latest news on SPIR and new OpenCL versions see:
 - <http://www.khronos.org/opencl/>

Resources: <https://www.khronos.org/opencl/>



OpenCL Programming Guide:
Aaftab Munshi, Benedict Gaster, Timothy G. Mattson and James Fung, 2011



Heterogeneous Computing with OpenCL
Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry and Dana Schaa, 2011

Other OpenCL resources

- New OpenCL user group
 - <http://comportability.org>
 - Forums
 - Downloaded examples
 - Training
 - Launched SC'12 in November
 - ACTION: register and become part of the community!!



PORTING CUDA TO OPENCL

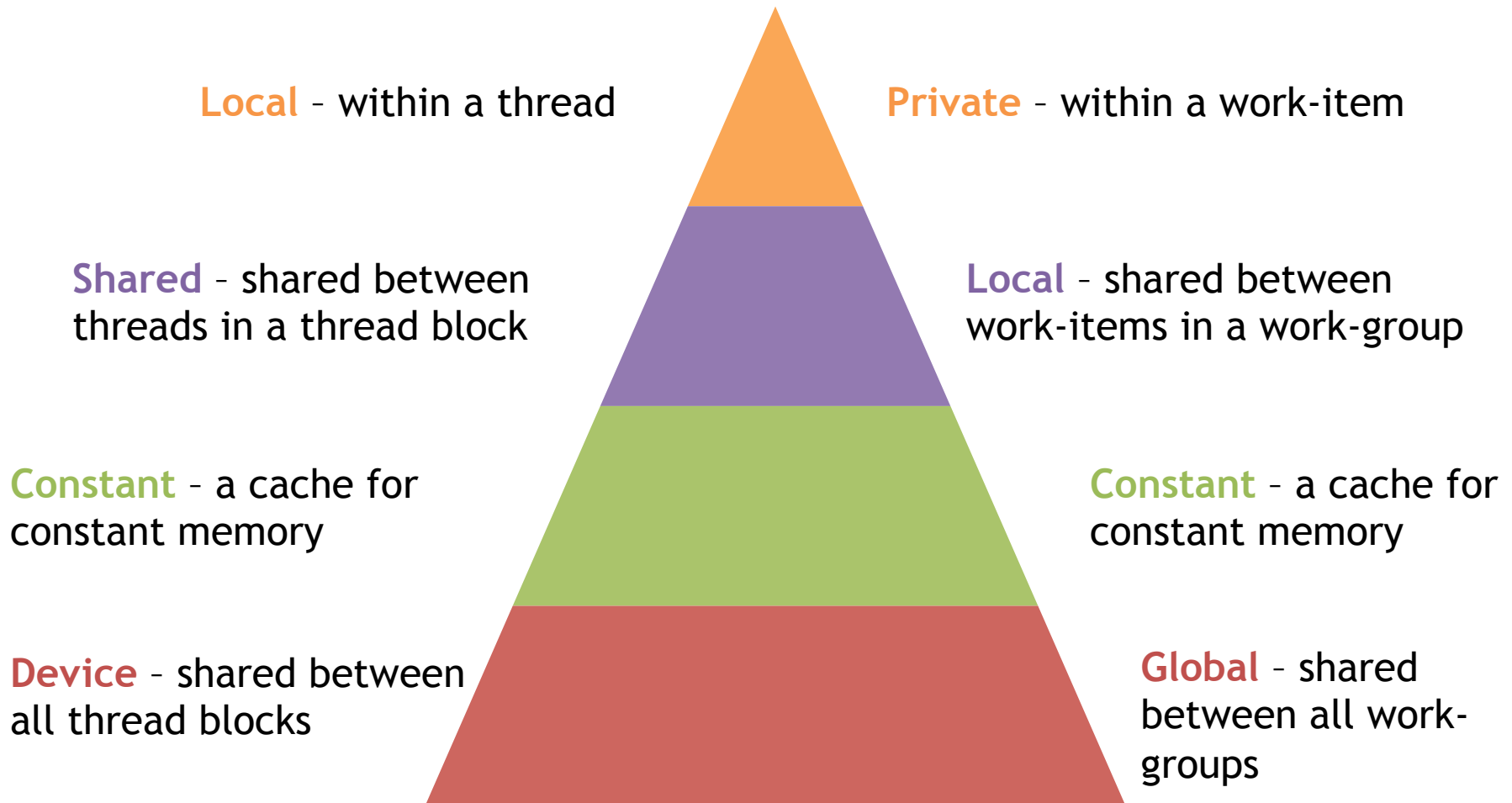
Introduction to OpenCL

- You've already done the hard work!
 - I.e. working out how to split up the problem to run effectively on a many-core device
- Switching between CUDA and OpenCL is mainly changes to host code syntax
 - Apart from indexing and naming conventions in kernel code (simple!)

Memory Hierarchy

CUDA

OpenCL



Allocating and copying memory

CUDA C

OpenCL C++

Allocate

```
float* d_x;  
cudaMalloc(&d_x,  
           sizeof(float)  
           *size);
```

```
cl::Buffer  
    d_x(begin(h_x), end  
        (h_x), true);
```

Host to Device

```
cudaMemcpy(d_x, h_x,  
           sizeof(float)  
           *size,  
  
           cudaMemcpyHostToDevice  
           );
```

```
cl::copy(begin(h_x), end  
        (h_x),  
  
        d_x);
```

Device to Host

```
cudaMemcpy(h_x, d_x,  
           sizeof(float)  
           *size,
```

```
cl::copy(d_x,  
        begin(h_x), end  
        (h_x));
```

Third party names are the property of their owners.

Declaring dynamic local/shared memory

CUDA C

1. Define an array in the kernel source as extern

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

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

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

OpenCL C++

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

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

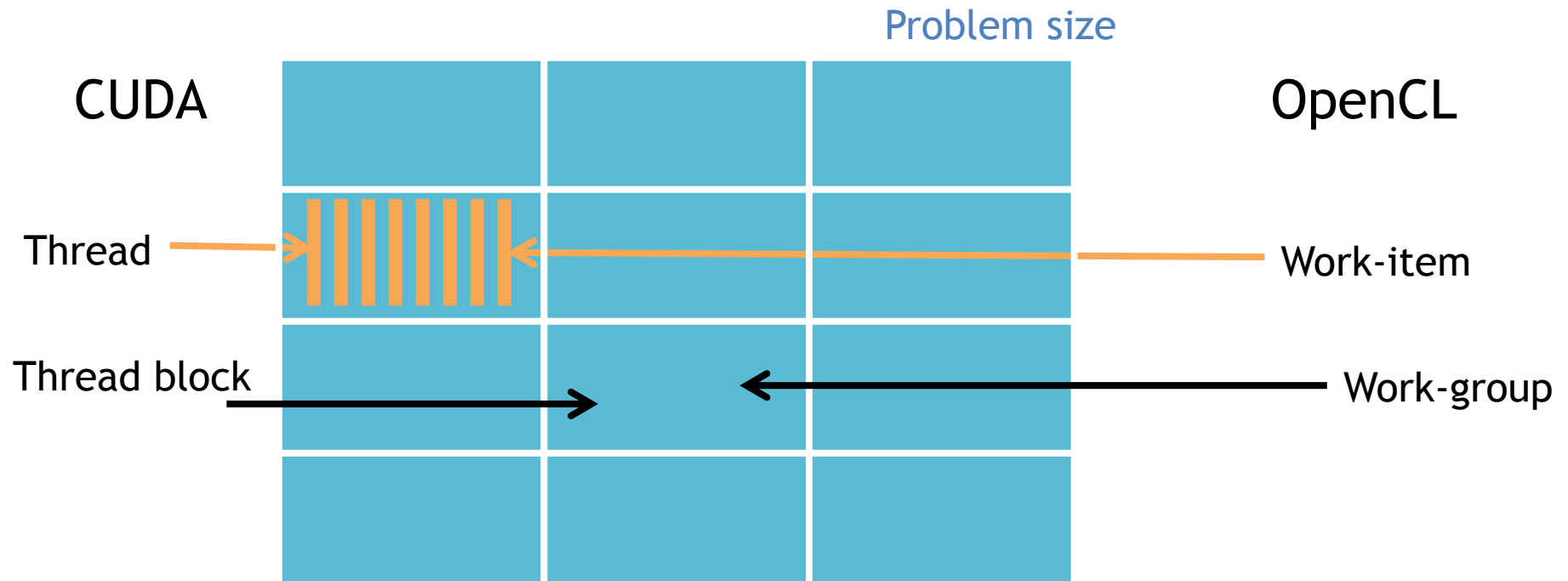
2. Define a local memory kernel kernel argument of the right size

```
cl::LocalSpaceArg localmem =  
    cl::Local(shared_mem_size);
```

3. Pass the argument to the kernel invocation

```
func(EnqueueArgs(...), localmem);
```

Dividing up the work



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

Enqueue a kernel

CUDA C

```
dim3 threads_per_block  
(30, 20);  
  
dim3 num_blocks(10, 10);  
  
kernel<<<num_blocks,  
    threads_per_block>>>();
```

OpenCL C++

```
const size_t global[2] =  
    {300, 200};  
  
const size_t local[2] =  
    {30, 20};  
  
kernel(EnqueueArgs(  
    NDRange(global),  
    NDRange(local), ...);
```

Indexing work

CUDA

gridDim

blockIdx

blockDim

gridDim * blockDim

threadIdx

blockIdx * blockDim + threadIdx

OpenCL

get_num_groups()

get_group_id()

get_local_size()

get_global_size()

get_local_id()

get_global_id()

Differences in kernels

- Where do you find the kernel?
 - OpenCL - a string (const char *), possibly read from a file
 - CUDA - a function in the host code
- Denoting a kernel
 - OpenCL - `__kernel`
 - CUDA - `__global__`
- When are my kernels compiled?
 - OpenCL - at runtime
 - CUDA - with compilation of host code

Host code

- By default, CUDA initializes the GPU automatically
 - If you needed anything more complicated (multi-card, etc.) you must do so manually
- OpenCL always requires explicit device initialization
 - It runs not just on NVIDIA® GPUs and so you must tell it which device to use

Thread Synchronization

CUDA

`__syncthreads()`

`__threadfenceblock()`

No equivalent

No equivalent

`__threadfence()`

OpenCL

`barrier()`

`mem_fence(
CLK_GLOBAL_MEM_FENCE |
CLK_LOCAL_MEM_FENCE)`

`read_mem_fence()`

`write_mem_fence()`

Finish one kernel and start
another

Translation from CUDA to OpenCL

CUDA	OpenCL
GPU	Device (CPU, GPU etc)
Multiprocessor	Compute Unit, or CU
Scalar or CUDA core	Processing Element, or PE
Global or Device Memory	Global Memory
Shared Memory (per block)	Local Memory (per workgroup)
Local Memory (registers)	Private Memory
Thread Block	Work-group
Thread	Work-item
Warp	No equivalent term (yet)
Grid	NDRange

More information

- <http://developer.amd.com/Resources/hc/OpenCLZone/programming/pages/portingcudatoopencl.aspx>

Exercise 5: CUDA and OpenCL

- **Goal:**
 - To understand CUDA and convert a serial matrix multiplication code into CUDA
- **Procedure:**
 - Examine the CUDA vadd program we have provided.
 - Using that program and these slides as your guide, convert provided matrix multiply program into a CUDA program (hint use the same algorithm as we used for OpenCL matrix multiply program).
- **Expected output:**
 - Test your answers and compare performance of the CUDA, OpenMP and serial programs.

SETTING UP OPENCL PLATFORMS

Some notes on setting up OpenCL

- We will provide some instructions for setting up OpenCL on your machine for a variety of major platforms and device types
 - AMD CPU, GPU and APU
 - Intel CPU
 - NVIDIA GPU
- We assume you are running 64-bit Ubuntu 12.04 LTS

Running OSX?

- OpenCL works out of the box!
- Just compile your programs with
`-framework OpenCL -DAPPLE`

Setting up with AMD GPU

- Install some required packages:
 - `sudo apt-get install build-essential linux-headers-generic debhelper dh-modaliases execstack dkms lib32gcc1 libc6-i386 opencl-headers`
- Download the driver from amd.com/drivers
 - Select your GPU, OS, etc.
 - Download the .zip
 - Unpack this with `unzip filename.zip`
- Create the installer
 - `sudo sh fglrx*.run --buildpkg Ubuntu/precise`
- Install the drivers
 - `sudo dpkg -i fglrx*.deb`
- Update your Xorg.conf file
 - `sudo amdconfig --initial --adapter=all`
- Reboot!
 - Check all is working by running `fglrxinfo`

* Fglrx is the name of AMD's graphics driver for the GPUs

Setting up with AMD CPU

- Download the AMD APP SDK from their website
- Extract with `tar -zxf file.tar.gz`
- Install with
 - `sudo ./Install*.sh`
- Create symbolic links to the library and includes
 - `sudo ln -s /opt/AMDAPP/lib/x86_64/* /usr/local/lib`
 - `sudo ln -s /opt/AMDAPP/include/* /usr/local/include`
- Update linker paths
 - `sudo ldconfig`
- Reboot and run `clinfo`

Third party names are the property of their owners.

Setting up with AMD APU

- The easiest way is to follow the AMD GPU instructions to install fglrx.
- This means you can use the CPU and GPU parts of your APU as separate OpenCL devices.
- You may have to force your BIOS to use integrated graphics if you have a dedicated GPU too.

Setting up with Intel CPU

- NB: requires an Intel® Xeon™ processor on Linux
- Download the Xeon Linux SDK from the Intel website
- Extract the download
 - `tar -zxvf download.tar.gz`
- Install some dependancies
 - `sudo apt-get install rpm alien libnuma1`
- Install them using alien
 - `sudo alien -i *base*.rpm *intel-cpu*.rpm
devel.rpm`
- Copy the ICD to the right location
 - `sudo cp /opt/intel/<version>/etc/
intel64.icd /etc/OpenCL/vendors/`

Setting up with Intel Xeon Phi

- Intel® Xeon Phi™ coprocessor are specialist processor only found in dedicated HPC clusters.
- As such, we expect most users will be using them in a server environment set up by someone else - hence we wont discuss setting up OpenCL on the Intel® Xeon Phi™ coprocessor in these slides

Setting up with NVIDIA GPUs

- Blacklist the open source driver (IMPORTANT)
 - `sudo nano /etc/modprobe.d/blacklist.conf`
 - Add the line: `blacklist nouveau`
- Install some dependencies
 - `sudo apt-get install build-essential linux-header-generic opencl-headers`
- Download the NVIDIA driver from their website and unpack the download
- In a virtual terminal (Ctrl+Alt+F1), stop the windows manager
 - `sudo service lightdm stop`
- Give the script run permissions then run it
 - `chmod +x *.run`
 - `sudo ./*.run`
- The pre-install test will fail - this is OK!
- Say yes to DKMS, 32-bit GL libraries and to update your X config
- Reboot!

C/C++ linking (gcc/g++)

- In order to compile your OpenCL program you must tell the compiler to use the OpenCL library with the flag: `-l OpenCL`
- If the compiler cannot find the OpenCL header files (it should do) you must specify the location of the CL/ folder with the `-I` (capital “i”) flag
- If the linker cannot find the OpenCL runtime libraries (it should do) you must specify the location of the lib file with the `-L` flag