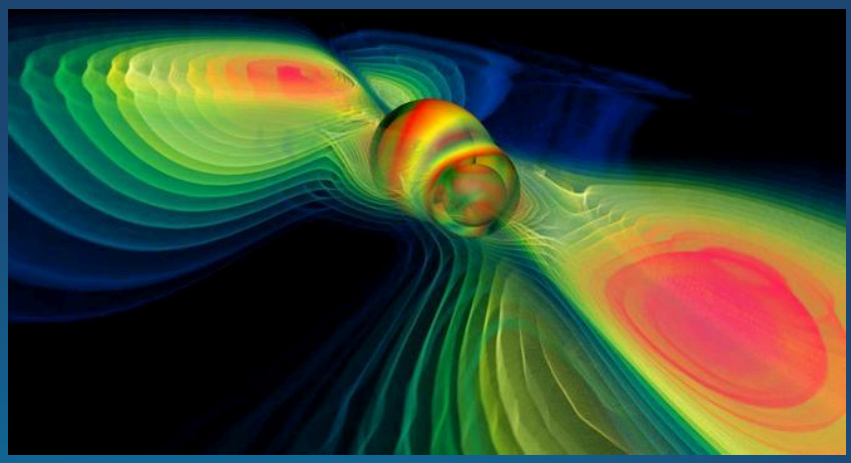
OpenCL Introduction

Frédéric Desprez
INRIA Grenoble Rhône-Alpes/LIG
Corse team



Simulation par ordinateur des ondes gravitationnelles produites lors de la fusion de deux trous noirs. Werner Benger, CC BY-SA

Acknowledgements

- Simon Mc Intosh-Smith (Univ. of Bristol)
- Tom Deakin (Univ. of Bristol)
- Brice Videau (CORSE, INRIA/LIG)
- Jean-François Méhaut (CORSE, INRIA/LIG)
- François Broquedis (CORSE, INRIA/LIG)
- Laura Grigori (ALPINES, INRIA)

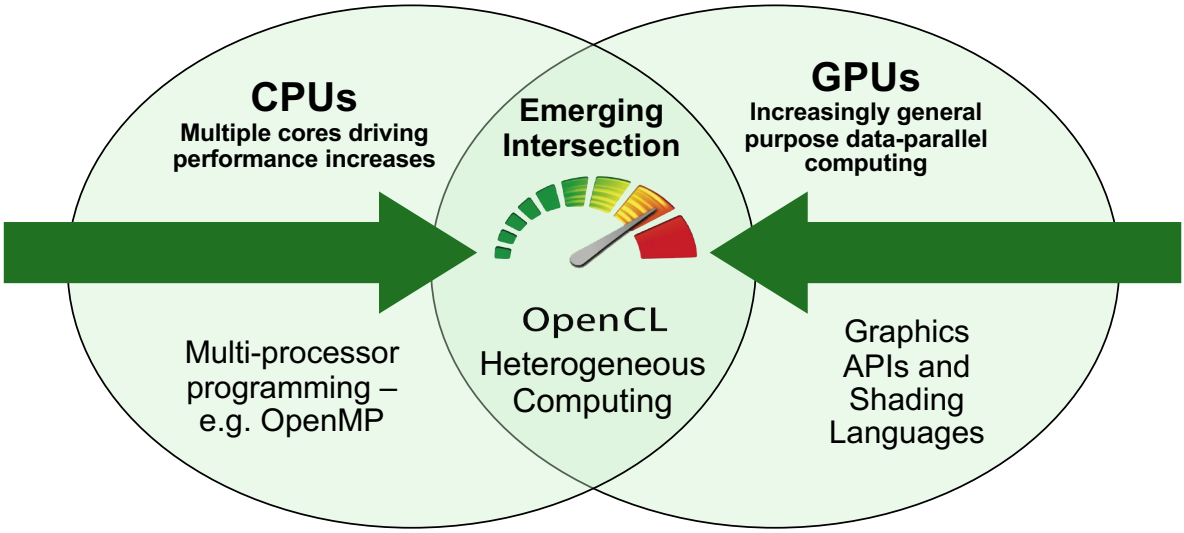
Agenda

- Introduction
- OpenCL generalities
- Vector addition example
- More on memory management in OpenCL
- Matrix multiplication example
- Optimization
- Other languages (CUDA, OpenACC)
- Conclusion

Today's Architectures and their Programming

- We now live in a heterogeneous world !
- Current parallel platforms include several different computing units
  - CPUs, GPUs, DSP processors, accelerators, FPGA processors, …
- Many different ways of programming these platforms
  - OpenMP, MPI, hybrid, thread libraries, computation or graphical libraries, ...
- Lack of (performance) portability

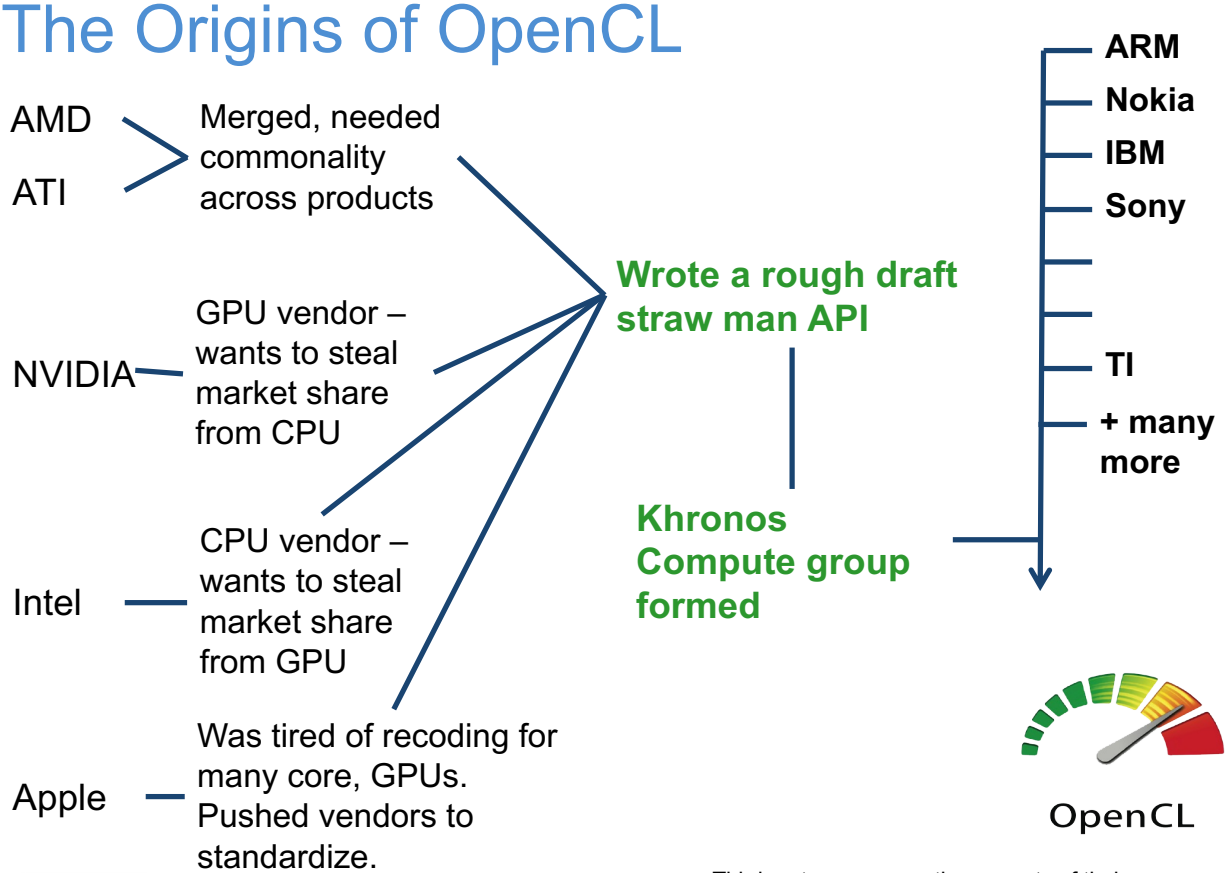
Industry Standards for Programming Heterogeneous Platforms



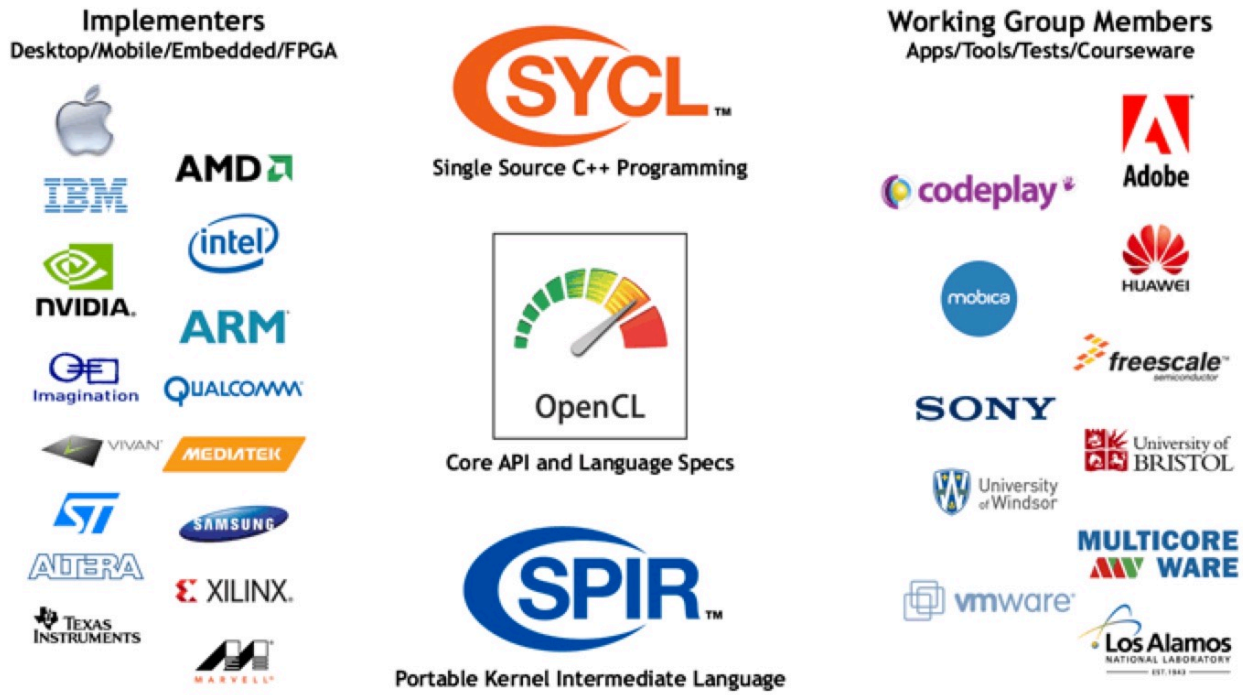
OpenCL – Open Computing Language

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

The Origins of OpenCL



OpenCL Ecosystem



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 on 1024x1024 = 1,048,576 kernel executions

Traditional loops

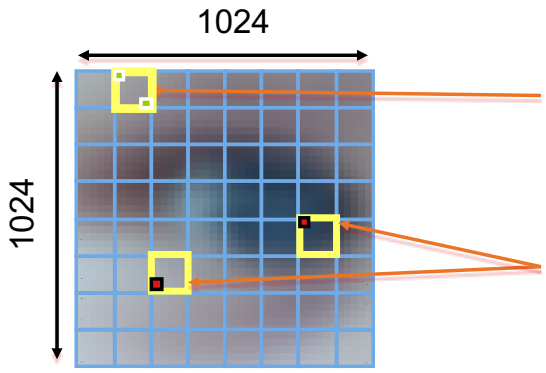
```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

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

A N-dimensional domain of work items

- **Global** dimensions
  - 1024x1024 whole problem space
- **Local** dimensions
  - 128x128 work-group, executes together
- Choice of the dimensions (1, 2, 3) that are “best” for your algorithm



Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

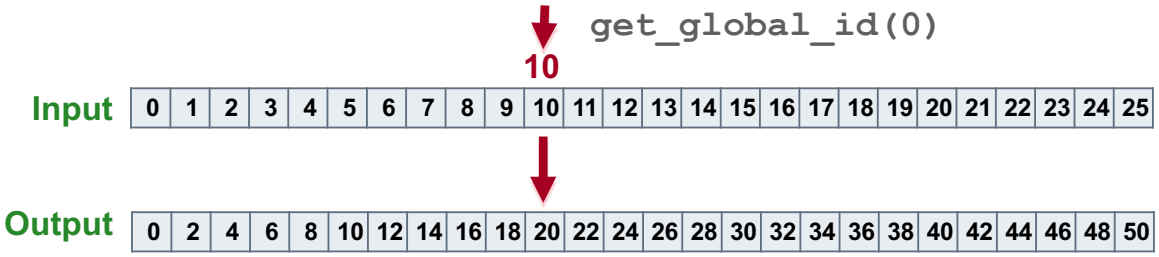
OpenCL N Dimensional Range (NDRange)

- The problem we want to compute should have some **dimensionality**
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension
  - This is called the **global** size
- We associate each point in the iteration space with a **work-item**
- Work-items are grouped into **work-groups**;
  - Work-items within a work-group can share **local memory** and can **synchronize**
- We can specify the number of work-items in a work-group
  - This is called the **local** (work-group) size
- Or the OpenCL run-time can choose the work-group size for you (usually not optimally)

Execution model (kernels)

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

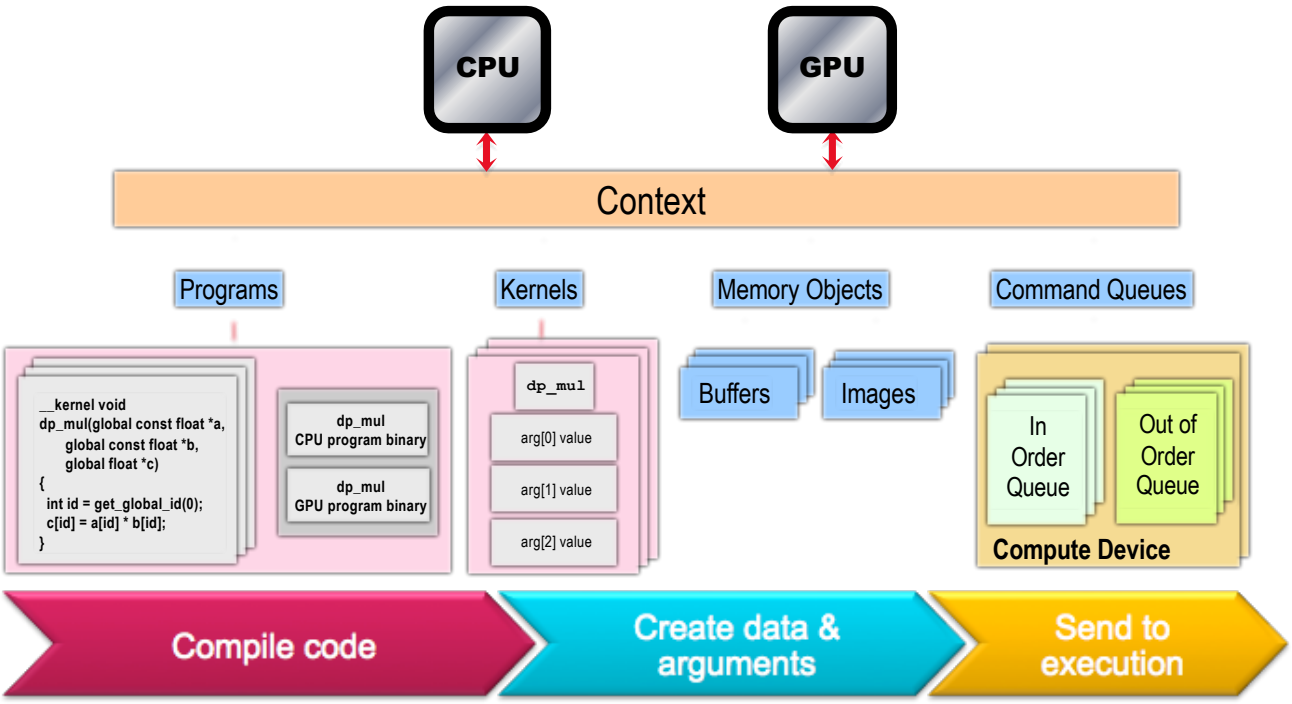
```
__kernel void times_two(
    __global float* input,
    __global float* output)
{
   int i = get_global_id(0);
   output[i] = 2.0f * input[i];
}
```



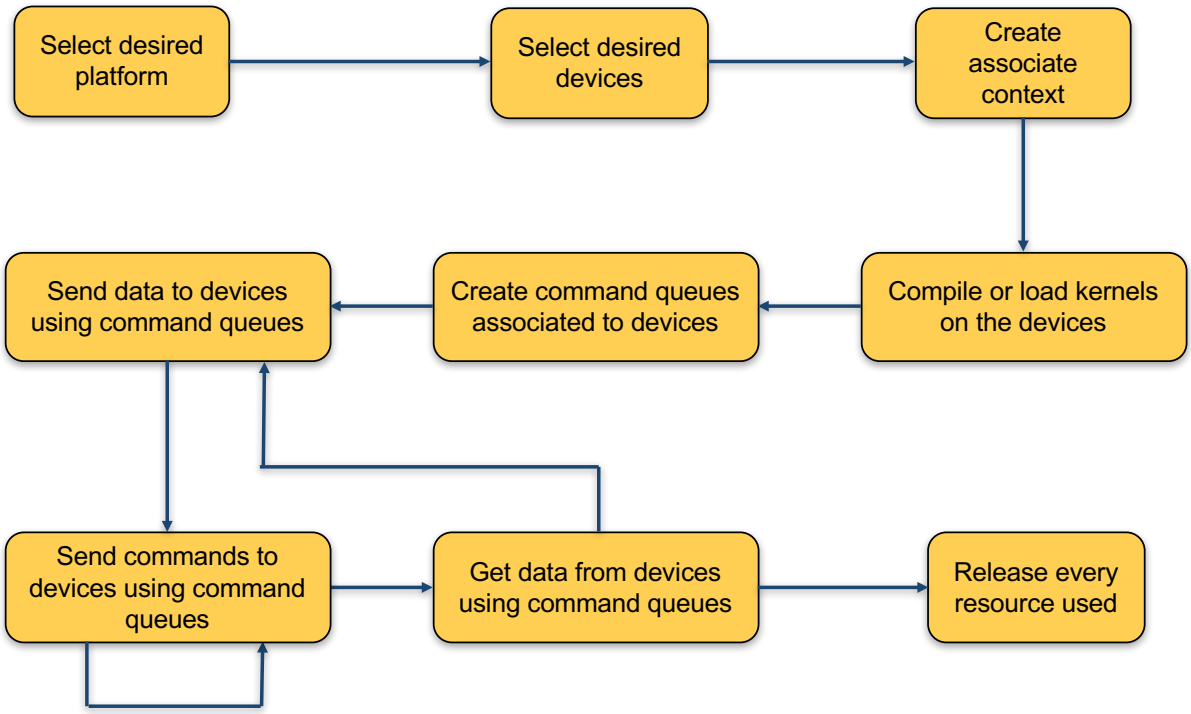
The OpenCL Specification

- **Platform model**
  - Specifies that one host is coordinating execution and one or more devices running OpenCL C kernels + abstract hardware model for devices
- **Execution model**
  - How the environment is configured by the host
  - How the host may direct the devices to perform work (concurrency model, work items and work groups)
- **Kernel programming model**
  - How the concurrency model is mapped to the physical hardware
- **Memory model**
  - Defines memory object types and abstract memory hierarchy, requirements for memory ordering and optional shared virtual memory between the host and devices

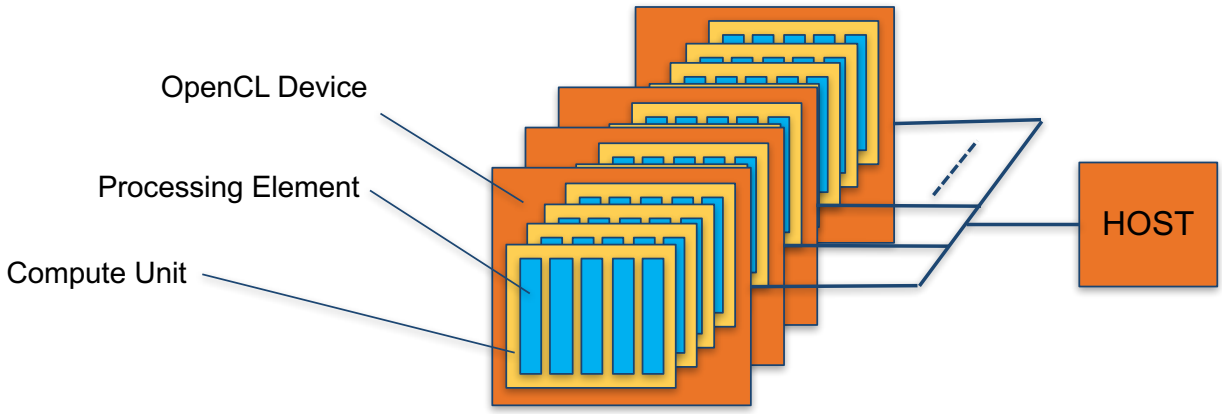
The basic platform and runtime APIs in OpenCL (using C)



General Workflow



OpenCL Platform Model



- One **host** and one (or more) OpenCL **device(s)**
- Devices are connected to the host
  - Devices are composed of one or mode **Compute Unit(s)**
  - Each compute device is divided into one or more **Processing Element(s)**
- Host issues **commands** to the devices
- Data are exchanged through **memory**
- Memory is divided into **host memory** and **device memory**

OpenCL Platform Example

One node, two CPU sockets, two GPUs

CPUs

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

GPUs

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

CU = Compute Unit; PE = Processing Element

Platform Selection

- **Select an appropriate platform for your application**

```
/* Get platforms */
cl_uint num_platforms;
clGetPlatformIDs( NULL, NULL, &num_platforms);
cl_platform_id *platforms = malloc(sizeof(cl_platform_id)*num_platforms);
clGetPlatformIDs( num_platforms, platforms, NULL);
/* … */
for (int i= 0; i<num_platforms; i++) {
        /* ... */
        clGetPlatformInfo(platform[i], CL_PLATFORM_VENDOR, ...);
        /* ... */
}
```

- **Several devices from the same vendors are common (graphic, computation)**

```
/* Get platforms (CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_CPU) */
cl_uint num_devices;
clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, NULL, NULL, num_devices);
cl_device_id *devices = malloc(sizeof(cl_device_id)*num_devices);
clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, num_devices, devices, NULL);
/* … */
for (int i= 0; i<num_devices; i++) {
        /* ... */
        clGetDeviceInfo(devices[i], CL_DEVICE_NAME, ...);
        /* ... */
}
```


Execution Model

- **Context**
  - Abstract environment within which coordination and memory management for kernel execution is valid and well defined
  - It coordinates the mechanisms for host-device interaction, manages the memory objects available to the devices, keep track of the program and kernels created for each device
  - It includes
    - One or more devices
    - Device memory
    - One or more command queues

```
/* Create Context */
cl_context_properties properties [] =
   {CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0};
cl_device_id = devices[] = {device_id_1, device_id_2};
cl_context context = clCreateContext( properties, 2, devices, NULL, NULL, NULL);

/* Create Context and skipping device selection */
cl_context_properties properties [] =
  {CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0};
cl_context context =
   clCreateContextFromType( properties, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
```

Execution Model

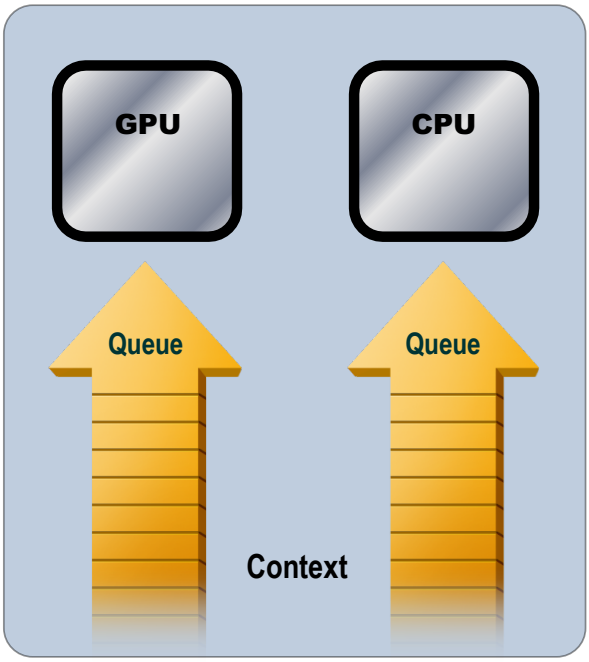
- **Command queues**
  - Commands sent from the host to the device
    - kernel executions, memory object management, synchronization
  - Communication mechanism used by the host to request an action from a device
  - Multiple queues can feed a single device (independent streams of commands)
  - Command queues can be
    - synchronous or asynchronous
    - event driven

```
/* Create Command Queues */
cl_command_queue queue =
    clCreateCommandQueue( context, devices[chosen_device], option, NULL);

/* option = bit field
   0, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, CL_QUEUE_PROFILING_ENABLE */
```

  - In OpenCL 2.0, possibility of having **device-side enqueuing**
    - device-side command-queue (out-of-order command queues)
    - parent and child kernels

Running on the CPU and GPU

- Kernels can be run on multiple devices at the same time
- We can exploit many GPUs and the host CPU for computation
- Simply define a context with multiple platforms, devices and queues
- We can even synchronize between queues using Events
- Can have more than one context



Create and Build the Program

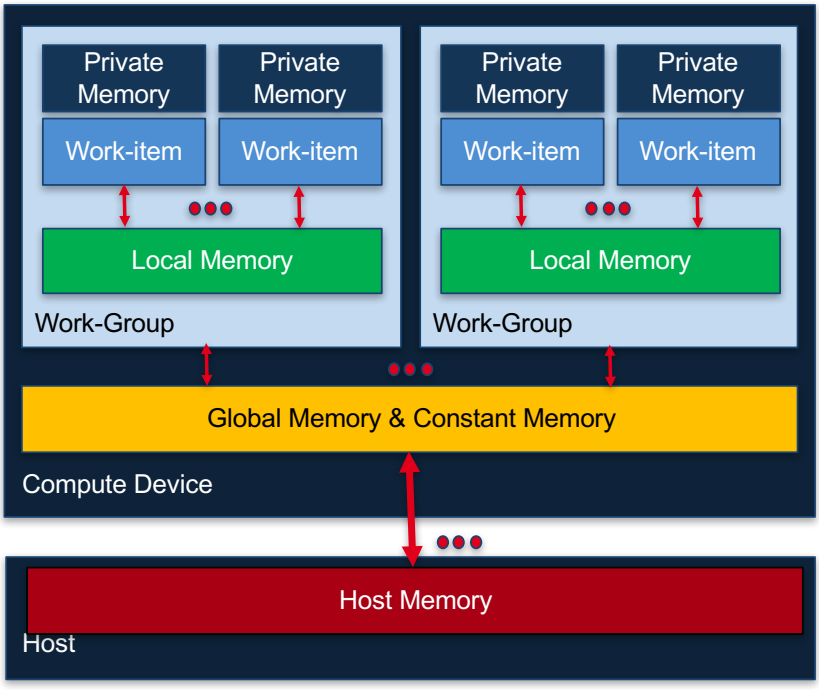
- A program object includes
  - A context
  - The program kernel source or binary
  - A list of target devices and build options
- Source code = string literal of read from a file

```
/* Building program */
cl_program program = clCreateProgramWithSource( context, string_count,
  strings, NULL, NULL);

/* if device_list is NULL, program is built for all available devices
   in the context */
clBuildProgram( program, num_devices, device_list, options, NULL, NULL);

cl_kernel kernel = clCreateKernel( program, "kernel_name", NULL);
```

- Kernels are extracted from the built program using their name
- OpenCL uses runtime compilation

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Share within a work-group
- **Global Memory**
  - Device RAM
  - Visible to all work-group
- **Constant Memory**
  - Cached Global memory
  - Visible to all work-group
- **Host memory**
  - On the CPU



Memory management is **explicit !**

- Programmer is responsible for moving data around Host → global → local *and* back

Types of memory objects

- Memory objects are used to move data back and forth with a device
  - Handles to a reference-counted regions of global memory
- **Buffers**
  - Equivalent of arrays in C,
  - Stored contiguously in memory
  - `clCreateBuffer`
- **Images**
  - Allows the hardware to take advantage of spacial locality and to use the hardware acceleration available on many devices
  - Dedicated functions to manipulate images
  - Opaque structure
  - `clCreateImage`
- **Pipes**
  - Ordered sequence of data items, FIFO based storage
  - Opaque structure
  - `clCreatePipe`

Buffer Creation and transfer

- Buffers are explicitly managed
  - Tied to a context

From the point of view of the device

```
/* Creating simple buffers */
cl_mem read_buffer = clCreateBuffer ( context, CL_MEM_READ_ONLY,
  buffer_size, NULL, NULL);

cl_mem write_buffer = clCreateBuffer ( context, CL_MEM_WRITE_ONLY,
  buffer_size, NULL, NULL);
```

- Transfering data

CL_TRUE = Blocking
CL_FALSE = Non-blocking

```
clEnqueueWriteBuffer( queue, read_buffer, CL_TRUE, 0, buffer_size,
  data_in, 0, NULL, NULL);

/* Processing that reads read_buffer and writes write_buffer */
/* … */

clEnqueueReadBuffer( queue, write_buffer, CL_TRUE, 0, buffer_size,
  data_out, 0, NULL, NULL);
```

Pinned Buffer Creation

- Pinned buffer creation can offer premium performance

```
cl_mem pinned_read_buffer =
   clCreateBuffer ( context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
         buffer_size, NULL, NULL);

cl_mem pinned_write_buffer =
   clCreateBuffer ( context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY,
         buffer_size, NULL, NULL);

unsigned char *data_in = clEnqueueMapBuffer( queue, pinned_read_buffer,
         CL_TRUE, CL_MAP_WRITE, 0, buffer_size, 0, NULL, NULL, NULL);

unsigned char *data_out = clEnqueueMapBuffer( queue, pinned_write_buffer,
         CL_TRUE, CL_MAP_READ, 0, buffer_size, 0, NULL, NULL, NULL);
```


Performing calculations

- Kernel usage

```
/* Place kernel parameters  in the kernel structure */
clSetKernelArg ( kernel, 0, sizeof(data_size), (void*)&data_size);
clSetKernelArg ( kernel, 1, sizeof(read_buffer), (void*)&read_buffer);
clSetKernelArg ( kernel, 2, sizeof(write_buffer), (void*)&write_buffer);

/* Enqueue a 1 dimensional kernel with a local size of 32 */
size_t localWorkSize[] = {32};
size_t globalWorkSize[] = { shrRoundUp(32, data_size) };
clEnqueueNDRangeKernel( queue, kernel, 1, NULL,
     globalWorkSize, localWorkSize, 0, NULL, NULL);
```

Event management

• Almost all functions presented end with

```
…, 0, NULL, NULL);
```

• These 3 arguments are used for event management, and thus asynchronous queue handling.

• Functions can wait for a number of events and can generate 1 event

```
event_t event_list[] = {event1, event2};
event_t event;
clEnqueueReadBuffer(queue, write_buffer, CL_FALSE, 0,
                    buffer_size, data_out, 2, event_list, event);
```

• Previous buffer read waits for 2 events and generates a third one that will happen when the read is completed

Release resources

• To exit cleanly from an OpenCL program, you have to free your resources

- Buffers `clReleaseMemObject`
- Kernel `clReleaseKernel`
- Events `clReleaseEvent`
- Programs `clReleaseProgram`
- Queues `clReleaseCommandQueue`
- Contexts `clReleaseContext`
- …

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

Vector addition

- Program to add two vectors

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

- Two parts for the OpenCL solution
  - Kernel code
  - Host code

- **Kernel**

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

Vector Addition – Host

- **The 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 functions)
  5. Submit ***commands*** … transfer memory objects and execute kernels

1. Define the platform

- Grab the first available platform

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

- Use the first CPU device the platform provides

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

- Create a simple context with a single device

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

- Create a simple command-queue to feed our device

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

2. Create and Build the program

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

• Build the program object:

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

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

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


3. Setup Memory Objects

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

• Create input vectors and assign values on the host

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

• Define OpenCL memory objects

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

4. Define the kernel

- Create kernel object from the kernel function “vadd”

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

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

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


5. Enqueue commands

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

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

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

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

5. Enqueue commands

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

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


Vector Addition – Host Program

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

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);

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

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

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

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

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

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

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

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

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

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

Vector Addition – Host Program

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

// get the list of GPU devices associated with context
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

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

Define platform and queues

```
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
memobjs[1
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);
```

Define memory objects

```
// create the program
program = cl
```

Create the program

```
// build t
err = clBu                                        LL,NULL);
```

Build the program

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

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

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

Create and setup kernel

```
// execute ke
err = clEnque                                     nel, 1, NULL,
                      global_work_size, NULL,0,NULL,NULL);
```

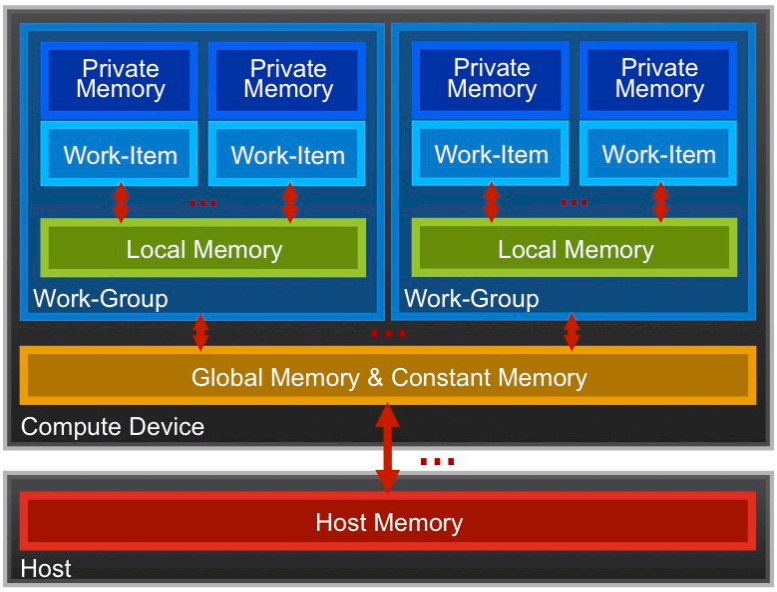
Execute the kernel

```
// read output array
err = clE                                         ],
                                                  dst,
                                  0, NULL, NULL);
```

Read results on the host

OpenCL Memory model

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

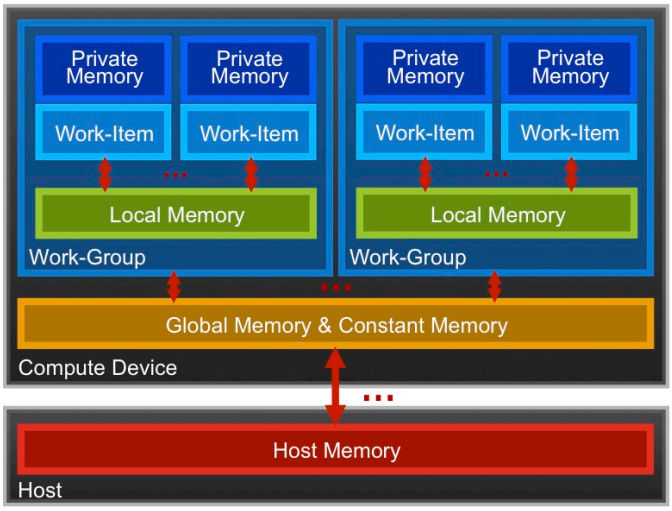


Memory management is **explicit**

You are responsible for moving data from host → global → local *and* back

OpenCL Memory model

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



Memory management is **explicit**

O(1-10) Gbytes/s bandwidth to discrete GPUs for Host ←→ Global transfers

Private Memory

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

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

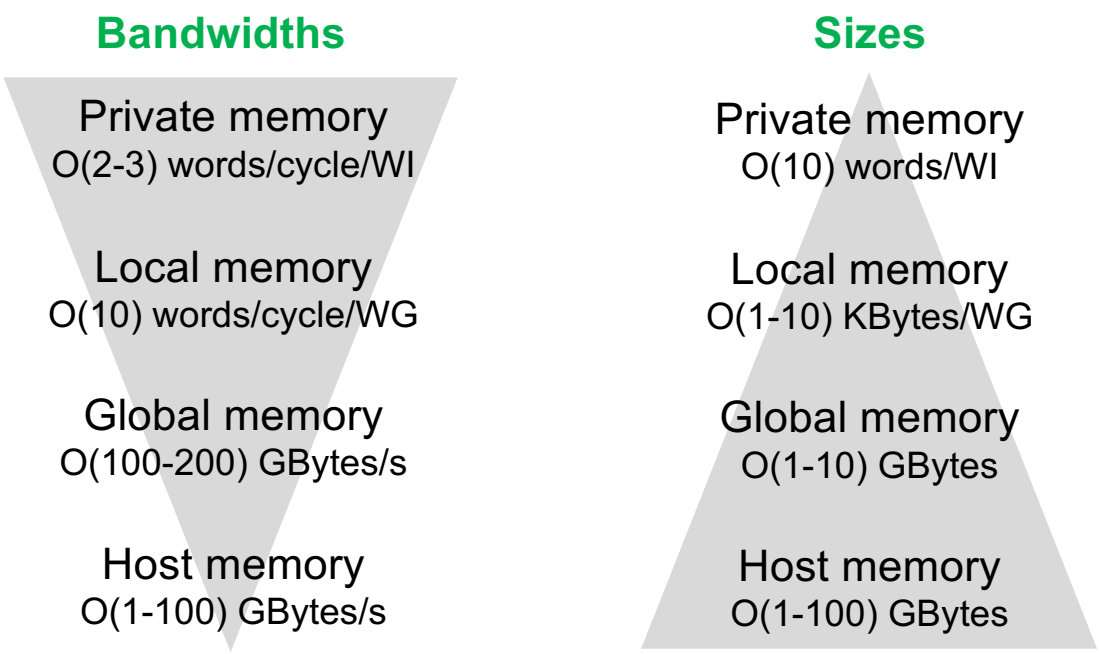
* Occupancy on a GPU

Local Memory*

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

* Typical figures for a 2013 GPU

The Memory Hierarchy



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

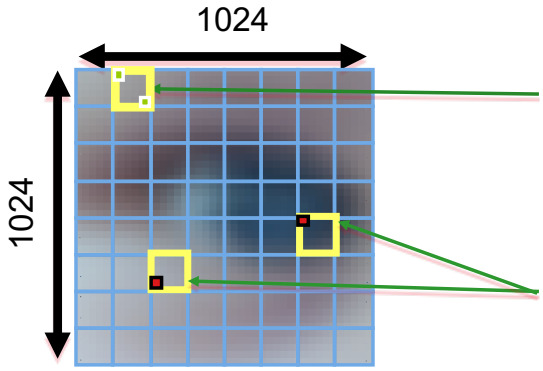
Memory Consistency

- 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)

Consider N-dimensional domain of work-items

- Global dimensions
  - 1024x1024 (whole problem space)
- Local dimensions
  - 64x64 (**work-group**, executes together)



Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

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

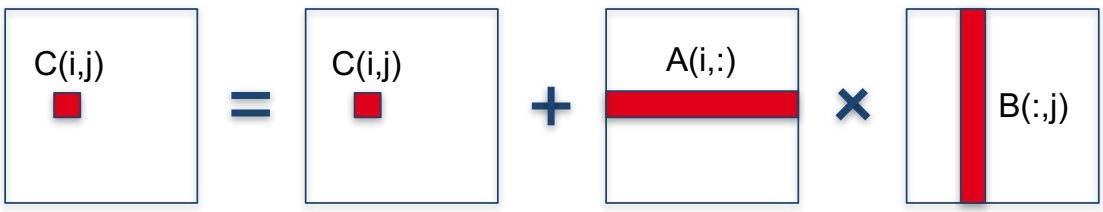
Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)

- **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

Matrix Product Sequential Code

Computation of C = A*B, dimA = N*P, dimB = P*M, dimC= N*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++) {
                // Dot product of a row of A and a column of B for each
                //element of C
                // c_{i,j} += a_{i,k} * b_{k,j}
                C[i*Ndim+j] += A[i*Ndim+k] * B[Pdim+j];
           }
      }
}
```



Matrix multiplication: sequential code


```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

We turn this into an OpenCL kernel!

Matrix multiplication: OpenCL kernel (1/2)


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

Mark as a kernel function and specify memory qualifiers

Matrix multiplication: OpenCL kernel (2/2)


```
__kernel void mat_mul(
  const int N,
  __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 < N; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Remove outer loops and set work-item coordinates

Matrix multiplication: OpenCL kernel


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

Matrix multiplication: OpenCL kernel improved

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

```
__kernel void mmul(const int N,
                   __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 < N; k++)
   tmp += A[i*N+k]*B[k*N+j];
  }
  C[i*N+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, h_A.begin(),h_A.end(), true);
  cl::Buffer d_b(context, h_B.begin(),h_B.end(), true);
  cl::Buffer d_c = cl::Buffer(context,
                              CL_MEM_WRITE_ONLY,
                              sizeof(float) * szC);


  cl::make_kernel<int, int, int,
                  cl::Buffer, cl::Buffer, cl::Buffer>
                  naive(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, h_C.begin(), h_C.end());

  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                    s in each matrix
  double start                    iming data
  cl::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);
```

Declare and initialize data

```
// Se                                    s,
// and
initma                                   ;
cl::Bu                                   end(), true);
cl::Bu                                   end(), true);
cl::Buffer d_c = cl::Buffer(context,
                            CL_MEM_WRITE_ONLY,
                            sizeof(float) * szC);
```

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

```
cl::make_kernel<int, int, int,
                cl::Buffer, cl::Buffer, cl::Buffer>
zero_mat(Ndim, Mdim, h_C);
start_time = wtime();
```

Create the kernel functor

```
// Compile for first kernel to setup program
program =                    lSource, true);
Context cc                   AULT);
cl::Comman
std::vecto
    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 platform and build program

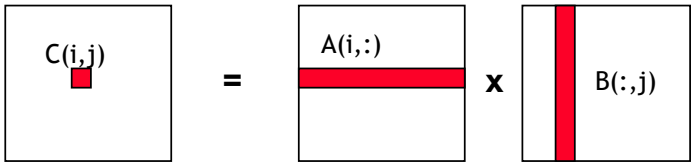
```
naive(cl::EnqueueArgs(queue,
                      cl::NDRange(Ndim, Mdim)),
      Ndi                            );

cl::copy                             .end());

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

Run the kernel and collect results

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

Optimizing matrix multiplication

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

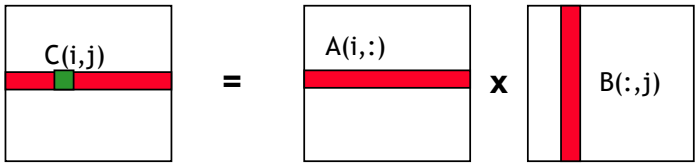


Dot product of a row of A and a column of B for each element of C

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

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



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

- And with an eye towards future optimizations, let's collect work-items into work-groups with 64 work-items per work-group

Matrix multiplication: One work item per row of C

```
__kernel void mmul(const int N,
  __global float *A, __global float *B, __global float *C)
{
  int j, k;
  int i = get_global_id(0);
  float tmp;
  for (j = 0; j < N; j++) {
   tmp = 0.0f;
   for (k = 0; k < N; k++)
     tmp += A[i*N+k]*B[k*N+j];
   C[i*N+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, h_A.begin(),h_A.end(), true);
  cl::Buffer d_b(context, h_B.begin(),h_B.end(), true);
  cl::Buffer d_c = cl::Buffer(context,
                              CL_MEM_WRITE_ONLY,
                              sizeof(float) * szC);


  cl::make_kernel<int, int, int,
                  cl::Buffer, cl::Buffer, cl::Buffer>
                  krow(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, h_C.begin(), h_C.end());

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


Matrix multiplication host program (C++ API)

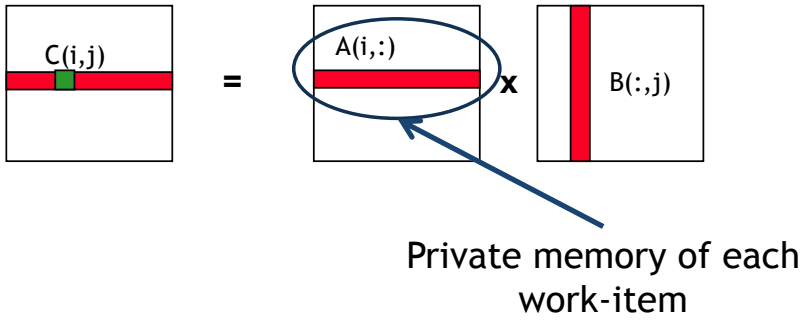
Changes to host program:

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(int
{
  std::vecto
  int Mdim, N
  int i, err
  int szA, sz
  double sta
  cl::Progra

  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";

                                                  true);
                                                  true);

                                                  C);

  cl::make_kernel<int, int, int,
                  cl::Buffer, cl::Buffer, cl::Buffer>
                  krow(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, h_C.begin(), h_C.end());

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

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: (Row of A in private memory)

Copy a row of A into private memory from global memory before we start with the matrix multiplications

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

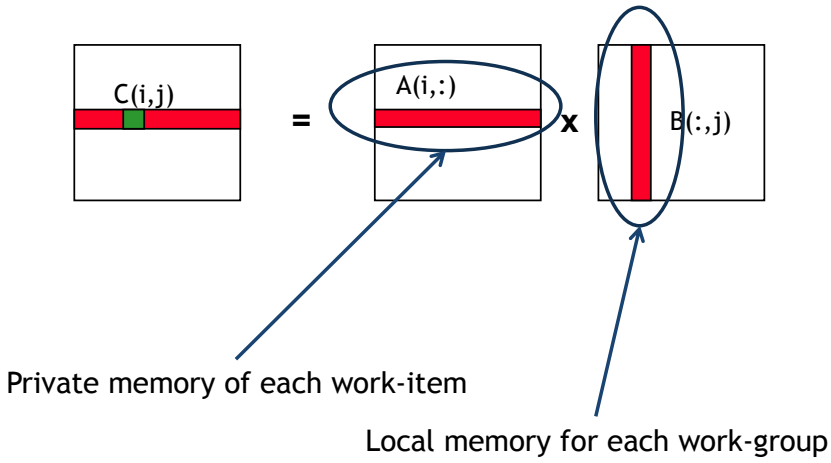
Setup a work array for A in private memory*

```
  for (k = 0; k < N; k++)
    Awrk[k] = A[i*N+k];

  for (j = 0; j < N; j++) {
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += Awrk[k]*B[k*N+j];

     C[i*N+j] += tmp;
  }
}
```

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

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)



Matrix multiplication: B column shared between work-items


```
__kernel void mmul(
      const int N,
    __global float *A,
    __global float *B,
    __global float *C,
    __local  float *Bwrk)
{
 int j, k;
 int i =
    get_global_id(0);

 int iloc  =
      get_local_id(0);

 int nloc =
      get_local_size(0);

 float tmp;
 float Awrk[1024];

 for (k = 0; k < N; k++)
   Awrk[k] = A[i*N+k];

 for (j = 0; j < N; j++) {

   for (k=iloc; k<N; k+=nloc)
     Bwrk[k] = B[k* N+j];

   barrier(CLK_LOCAL_MEM_FENCE);

   tmp = 0.0f;
   for (k = 0; k < N; k++)
     tmp += Awrk[k]*Bwrk[k];

   C[i*N+j] = tmp;

   barrier(CLK_LOCAL_MEM_FENCE);
 }
}
```

Pass a work array in local memory to hold a column of B. All the work-items do the copy "in parallel" using a cyclic loop distribution (hence why we need iloc and nloc)

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, h_A.begin(),h_A.end(), true);
  cl::Buffer d_b(context, h_B.begin(),h_B.end(), true);
  cl::Buffer d_c = cl::Buffer(context,
                              CL_MEM_WRITE_ONLY,
                              sizeof(float) * szC);

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

  cl::make_kernel<int, int, int,
            cl::Buffer, cl::Buffer, cl:::Buffer,
            cl::LocalSpaceArg>
               rowcol(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, h_C.begin(), h_C.end());

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


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
{
  s
  i                                                  (), true);
  i                                                  (), true);
  i
  d                                                  LY,
  cl::Program program;                               szC);

  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);

  cl::make_kernel<int, int, int,
            cl::Buffer, cl::Buffer, cl:::Buffer,
            cl::LocalSpaceArg>
               rowcol(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, h_C.begin(), h_C.end());

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

Making matrix multiplication *really* fast

• Our goal has been to describe how to work with private, local and global memory
• We've ignored many well-known techniques for making matrix multiplication fast
  - The number of work items must be a multiple of the fundamental machine "vector width".
    • This is the wavefront on AMD, warp on NVIDIA, and the number of SIMD lanes exposed by vector units on a CPU
  - To optimize reuse of data, you need to use blocking techniques
    • Decompose matrices into tiles such that three tiles just fit in the fastest (private) memory
    • Copy tiles into local memory
    • Do the multiplication over the tiles

Portable performance in OpenCL

• **Don't optimize too hard for any one platform, e.g.**
  - Don't write specifically for certain warp/wavefront sizes etc
  - Be careful not to rely on specific sizes of local/global memory
  - OpenCL's vector data types have varying degrees of support – faster on some devices, slower on others
  - Some devices have caches in their memory hierarchies, some don't, and it can make a big difference to your performance without you realizing
  - Choosing the allocation of Work-Items to Work-Groups and dimensions on your kernel launches
  - Some OpenCL SDKs give useful feedback about how well they can compile your code (but you have to turn on this feedback)

• It is a good idea to **try your code on several different platforms** to see what happens (profiling is good!)
  - At least two different GPUs (ideally different vendors) and at least one CPU

• **Auto-tuning !**

Some general issues to think about

- **Tiling size (work-group sizes, dimensionality etc.)**
  - For block-based algorithms (e.g. matrix multiplication)
  - Different devices might run faster on different block sizes
- **Data layout**
  - Array of Structures or Structure of Arrays (AoS vs. SoA)
  - Column or Row major
- **Caching and prefetching**
  - Use of local memory or not
  - Extra loads and stores assist hardware cache?
- **Work-item / work-group data mapping**
  - Related to data layout
  - Also how you parallelize the work
- **Operation-specific tuning**
  - Specific hardware differences
  - Built-in trig / special function hardware
  - Double vs. float (vs. half)

From Zhang, Sinclair II and Chien: Improving Performance Portability in OpenCL Programs – ISC13

OpenCL versus CUDA

- Compute Unifed Device Architecture (**CUDA**)
- Parallel computing architecture developed by NVIDIA for graphics processing and **GPU** programming

- **Pros**
  - Large range of hardware vs NVIDIA-only platforms
  - Higher level for OpenCL (a bit less complicated), lower level for CUDA (complicated but highly tuned code)

- **Cons**
  - CUDA has more mature tools, including a debugger and a profiler, also CUBLAS and CUFFT
  - Better performance maybe with CUDA (but with a lot of efforts)

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

OpenACC

- **OpenACC API**
  - Describes a collection of compiler directives to specify loops and regions of code in standard C, C++, and Fortran to be of loaded from a host CPU to an attached accelerator
  - Portability across operating systems, host CPUs, and accelerators.
  - Similar to OpenMP in terms of program annotation
- **Pros**
  - Simple to learn
  - Easy and incremental application to your code
  - Code can also be compiled to run on CPUs

Cons

- Give up control
- May be less efficient than optimized CUDA code
- Future support unclear

```
/* Vector addition */
#pragma acc kernels loop independent copyin(a[0:N], b[0:N]) copyout(c[0:N])
{
   for(int i = 0; i < N; i++){
        c[i] = a[i] + b[i];
   }
}
```

http://openacc.org

Conclusions

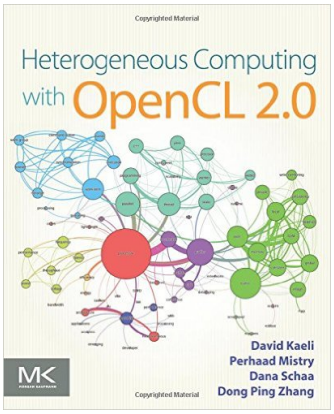
- OpenCL has widespread industrial support
- OpenCL defines a platform-API/framework for heterogeneous computing, not just GPGPU or CPU-offload programming
- OpenCL has the potential to deliver portably performant code; but it has to be used correctly
- The latest C++ and Python APIs make developing OpenCL programs much simpler than before
- The future is (quite) clear
  - OpenCL is one 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

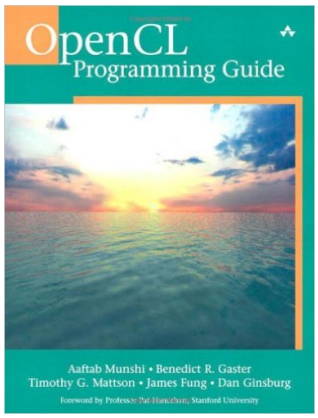
Some References

OpenCL reference card
https://www.khronos.org/files/opencl21-reference-guide.pdf

Heterogeneous Computing with OpenCL 2.0
David R. Kaeli, Perhaad Mistry, Dana Schaa, Dong Ping Zhan

OpenCL Programming Guide
Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg





http://www.khronos.org/opencl/

Frédéric Desprez
Frederic.Desprez@inria.fr

Inria
INVENTEURS DU MONDE NUMÉRIQUE