INSTITUT
de MATHEMATIQUES
de TOULOUSE

Introduction au calcul GPGPU

Partie 2 : Découverte d'openCL

Laurent Risser

Ingénieur de Recherche à l'Institut de Mathématiques de Toulouse

lrisser@math.univ-toulouse.fr

Présentation basée sur Hands On OpenCL

- https://github.com/HandsOnOpenCL/Lecture-Slides/releases
- Largement simplifiée et réorganisée pour introduire les concepts essentiels de la programmation GPGPU (initialement 275 slides)

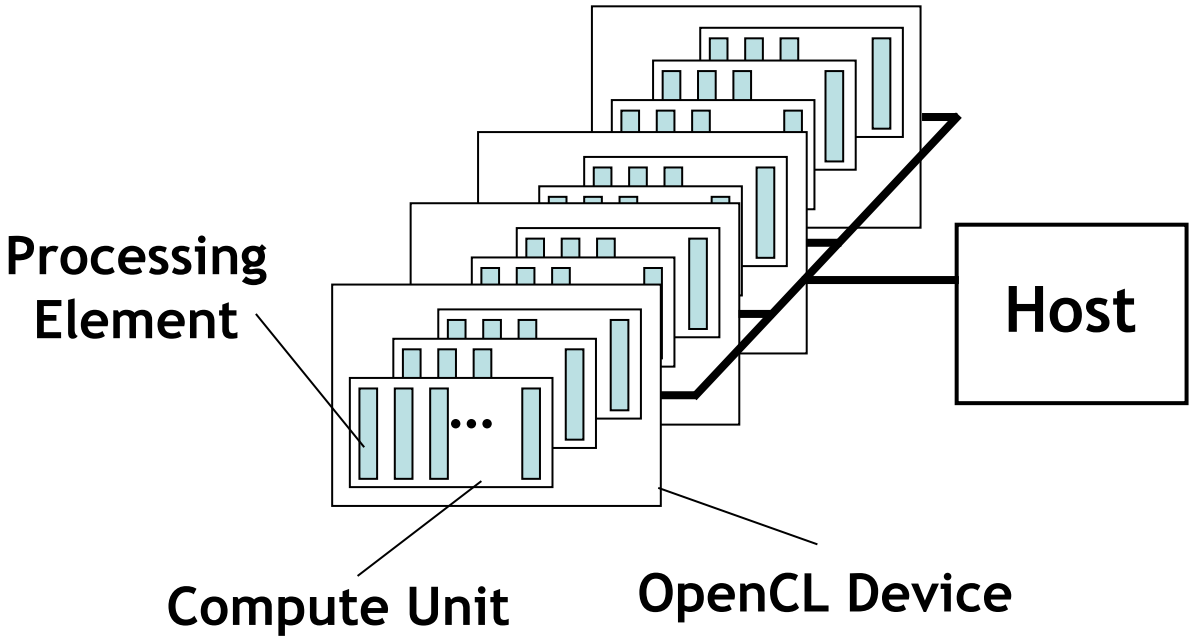
Plan de la présentation

1. Main concepts
2. The pyopencl API
3. Programming and optimizing openCL kernels
4. Switching between openCL and CUDA

Partie 1

MAIN CONCEPTS

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

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

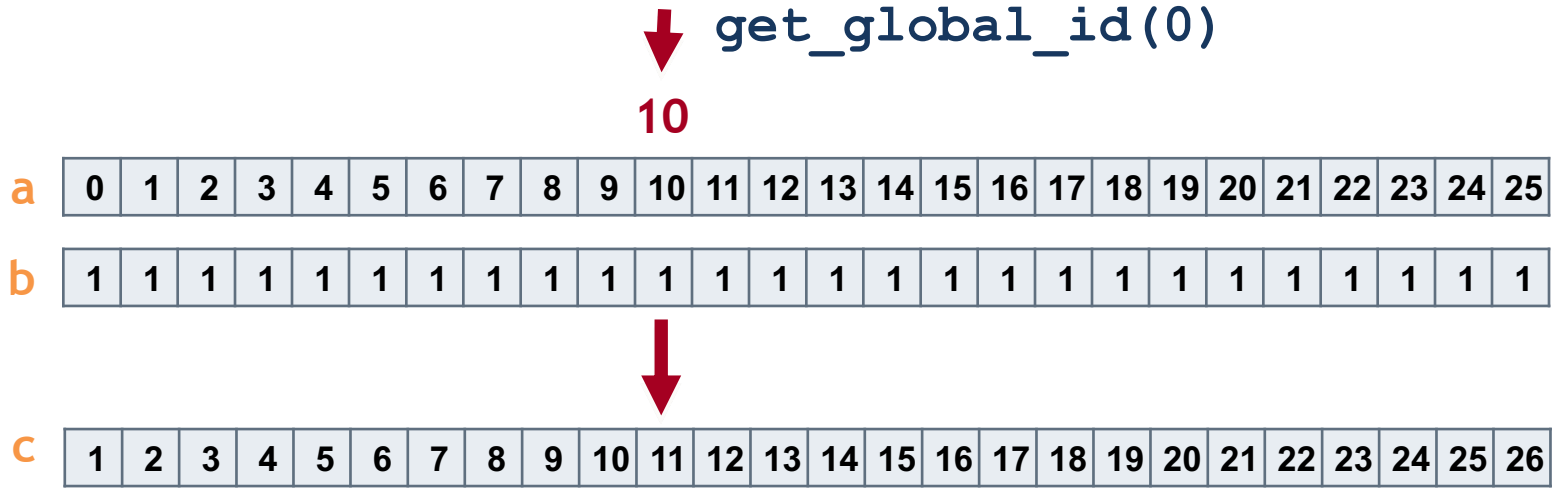
```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}

// many instances of the kernel,
// called work-items, execute
// in parallel
```

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
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

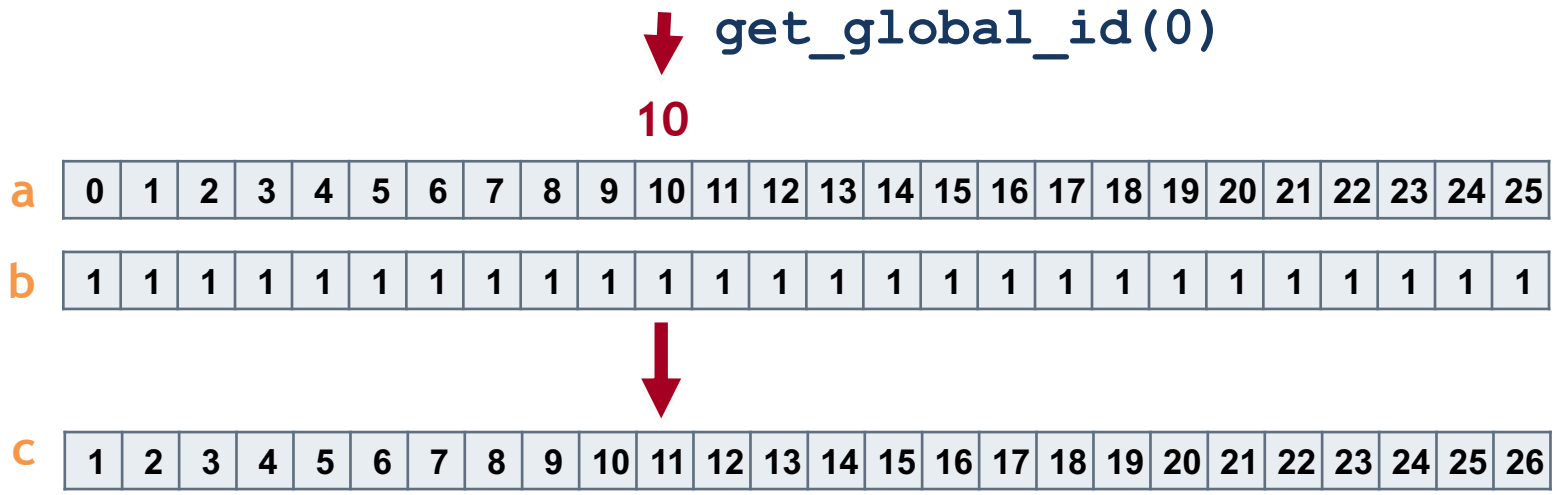


Execution model (kernels)

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

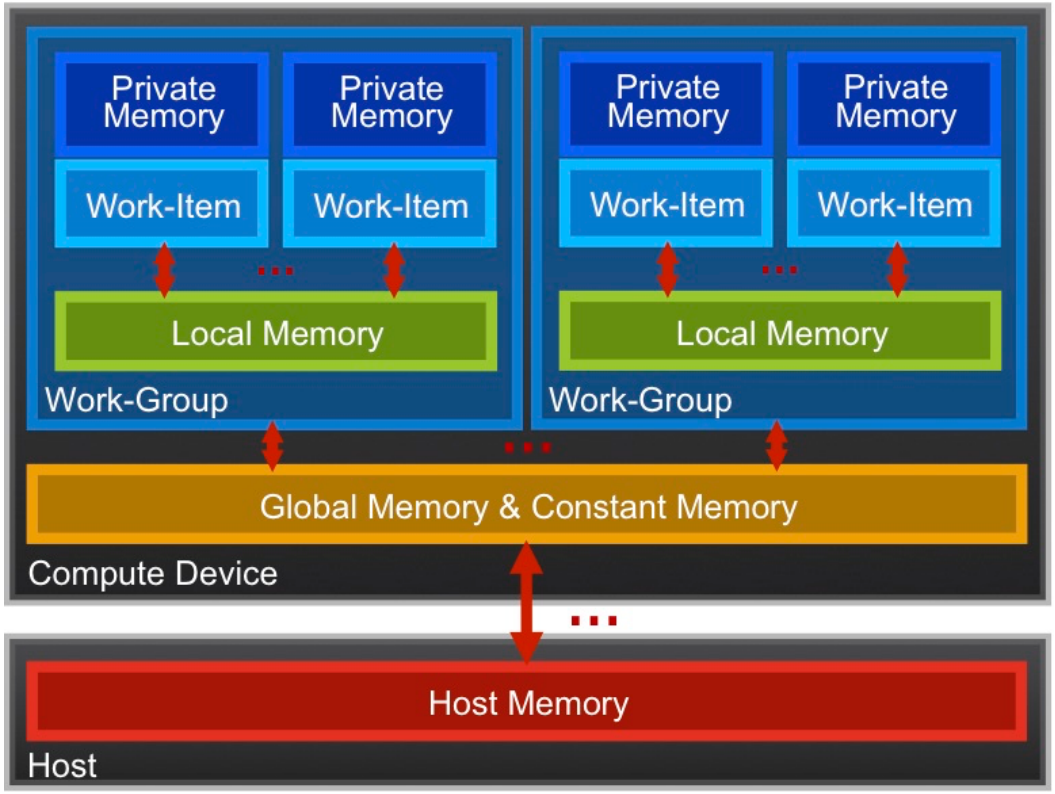
```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

- The problem we want to compute should have some dimensionality.
- When we execute the kernel we specify up to 3 dimensions.
- We associate each point in the iteration space with a work-item.



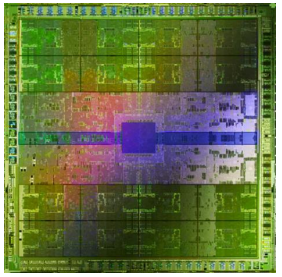
OpenCL Memory model

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



Complex memory model due to:

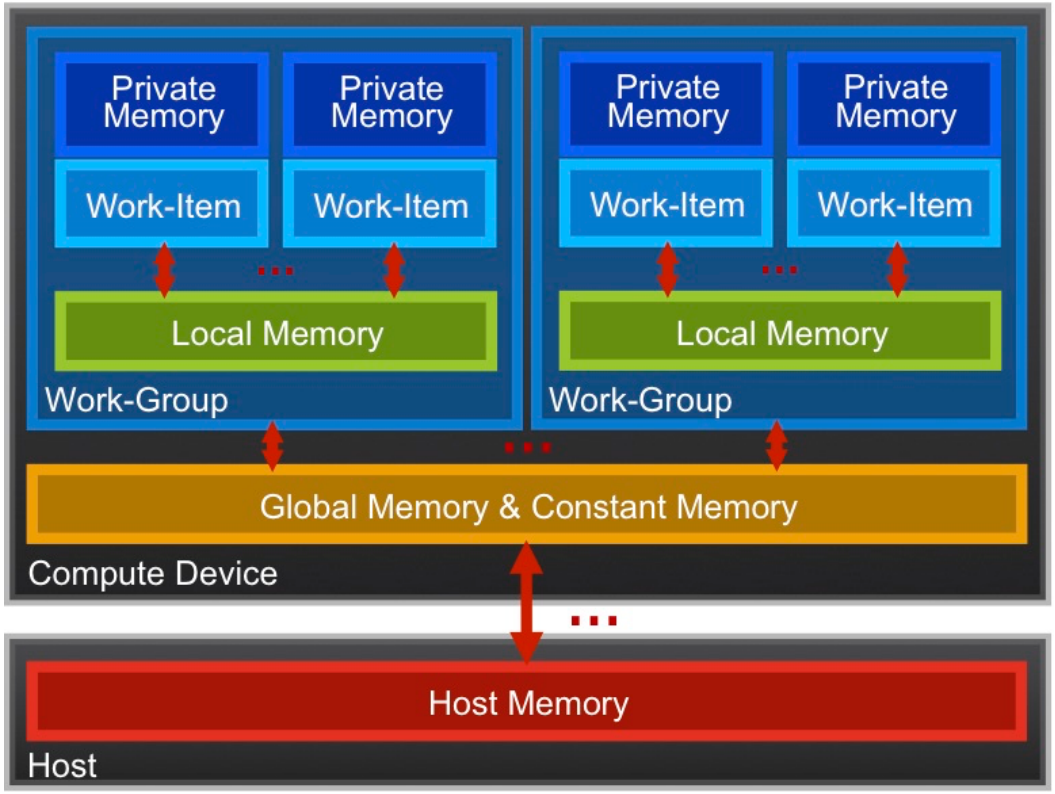
- Physical constrains in GPU processors architecture
- Cost/physical constrains on L1, L2, L3 caches



NVIDIA® Tesla® C2090

OpenCL Memory model

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

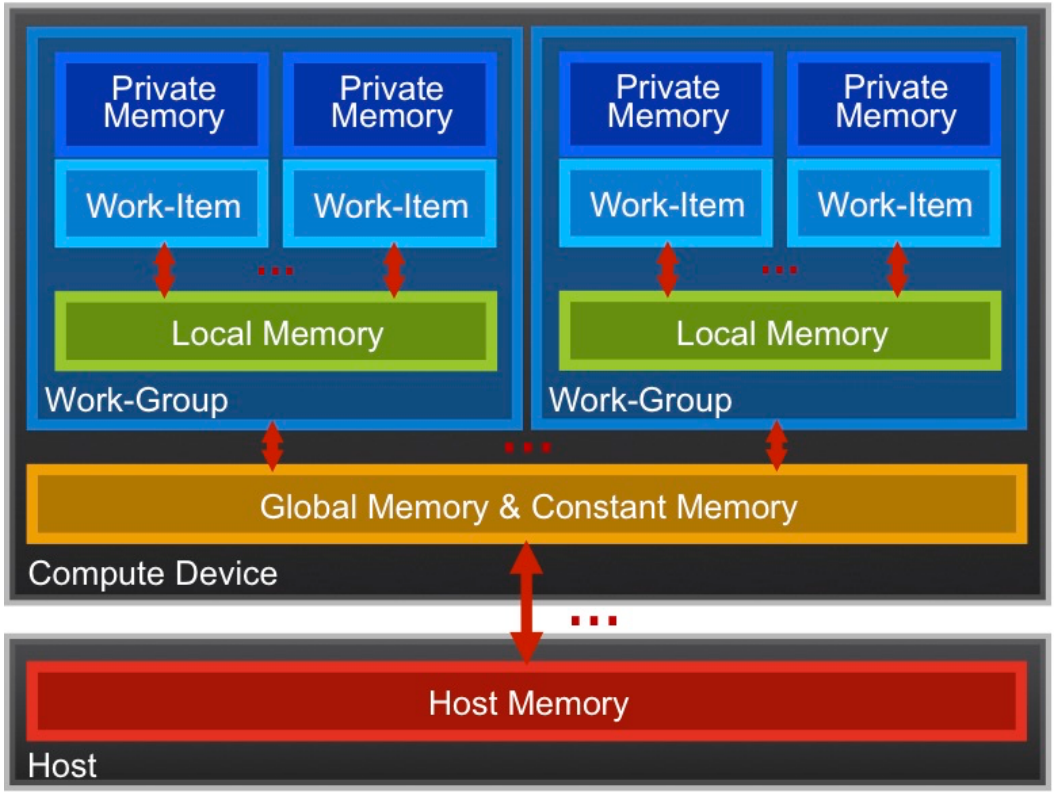


Explicit memory management:

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

OpenCL Memory model

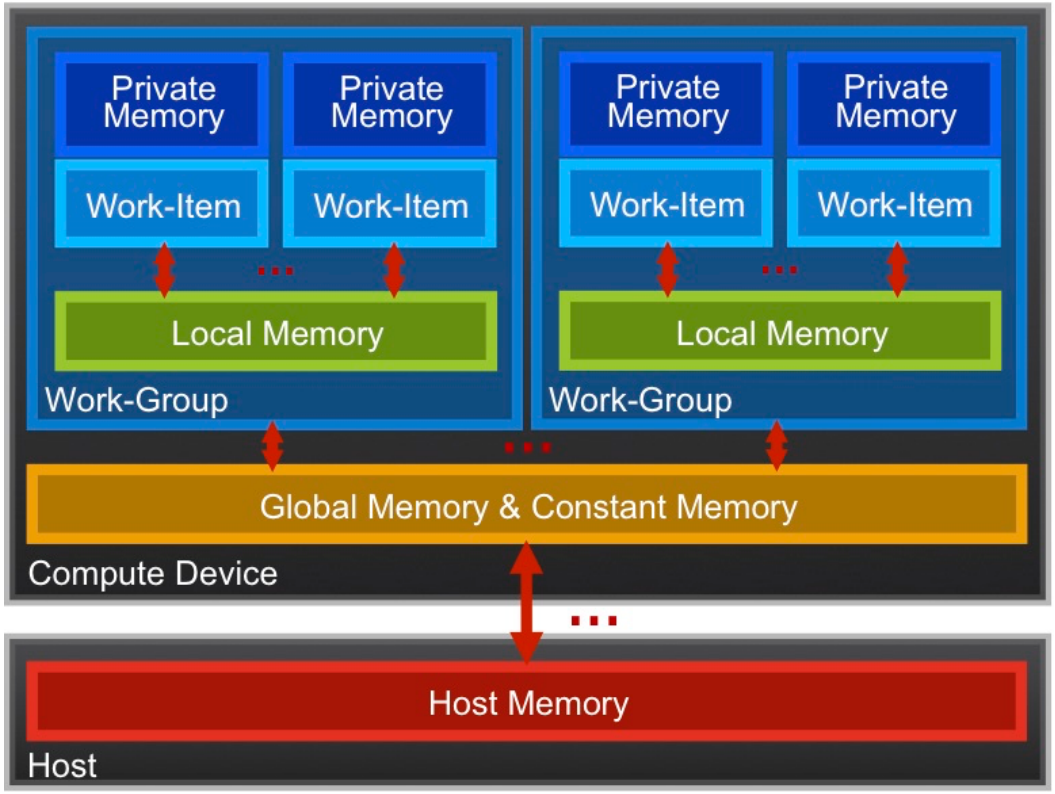
- *Private Memory*
  - Per work-item only
- *Local Memory*
  - Shared within a work-group only
- *Global Memory /Constant Memory*
  - Visible to all work-groups
- *Host memory*
  - On the CPU



- Work-items are grouped into work-groups;
- work-items within a work-group share local memory and can synchronize.
- We can specify the number of work-items in a work-group.
- OpenCL run-time can alternatively choose the work-group size for you (usually not optimally)

OpenCL Memory model

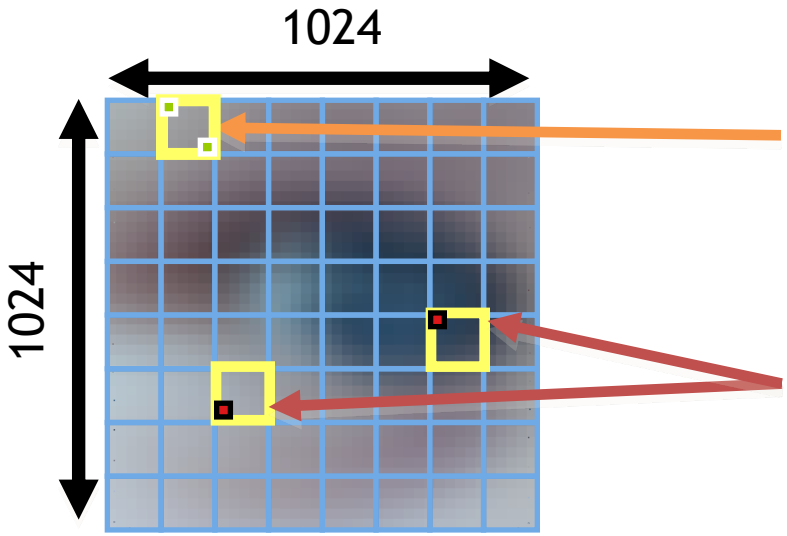
- *Private Memory*
  - Fastest & smallest: O(10) words/WI
- *Local Memory*
  - O(1-10) Kbytes per work-group
- *Global Memory /Constant Memory*
  - O(1-10) Gbytes of Global memory
  - O(10-100) Kbytes of Const. memory
- *Host memory*
  - On the CPU - GBytes



Remark: O(1-10) Gbytes/s bandwidth for Host mem. <-> GPU Global mem. transfers

Example: treatment of a 1024x1024 image

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



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

Cannot synchronize between work-groups within a kernel: Incremental use of small kernels.

Remark: The optimal choice of local dimensions depends on the processor (device) Properties. Auto-tuning strategies are however common.

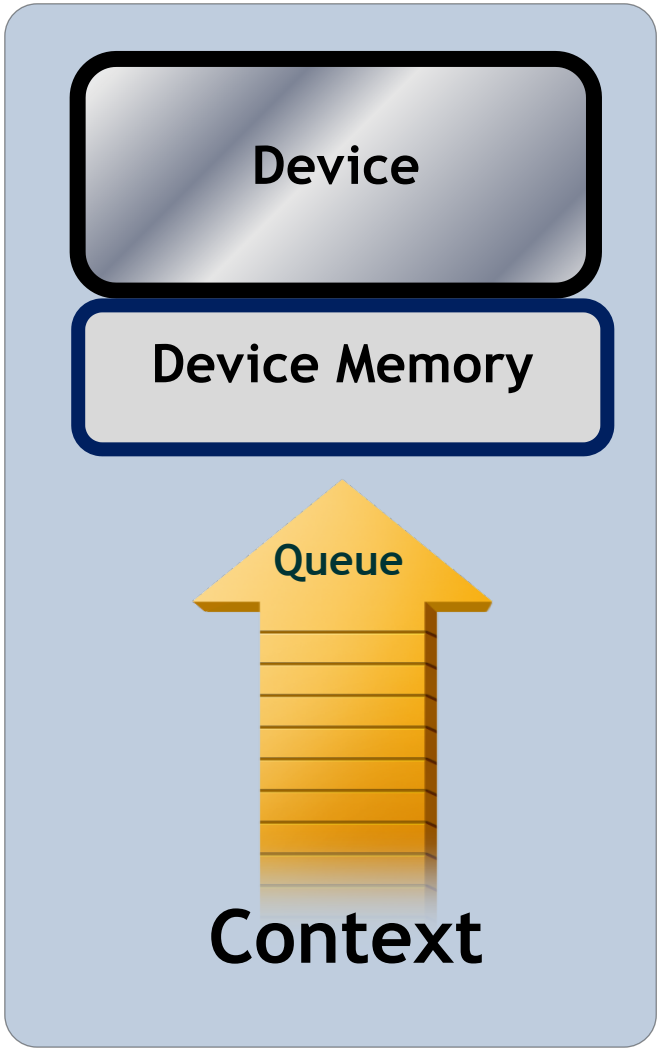
Context and Command-Queues

Context: 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

A *command-queue* manages all commands (kernel execution, synchronization, and memory transfer operations) for a single device within a context.



Part 2

THE PYOPENCL API

As seen in part 1, 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 (in OpenCL)
- Host code (in C, C++, Python, …)

… we focus here on the Python API: PyOpenCL

Vector Addition – Kernel program

```
__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 program

- The host program is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels

```
import pyopencl as cl
import numpy
# create context, queue and program
context = cl.create_some_context()
queue = cl.CommandQueue(context)
kernelsource = open('vadd.cl').read()
program = cl.Program(context, kernelsource).build()

# create host arrays
N = 1024
h_a = numpy.random.rand(N).astype(float32)
h_b = numpy.random.rand(N).astype(float32)
h_c = numpy.empty(N).astype(float32)

# create device buffers
mf = cl.mem_flags
d_a = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_a)
d_b = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_b)
d_c = cl.Buffer(context, mf.WRITE_ONLY, h_c.nbytes)

# run kernel and return results
vadd = program.vadd
vadd.set_scalar_arg_dtypes([None, None, None, numpy.uint32])
vadd(queue, h_a.shape, None, d_a, d_b, d_c, N)

cl.enqueue_copy(queue, h_c, d_c)
```

See 2.3

See 2.4

Build openCL kernels using pyopencl

Kernel source string can be defined with three quote marks *in the Python code*:

```
kernelsource =
    '''
    __kernel void func(){
        …
        }
    '''
```

Or in a file and loaded at runtime:

```
kernelsource = open('vadd.cl').read()
```

The program object is then created and built using:

```
program = cl.Program(context, kernelsource).build()
```

Run openCL kernels using pyopencl

Kernels can be called as a method of the built program object; as in

```
vadd(queue, h_a.shape, None, d_a, d_b, d_c, N)
```

More generally the command is:

program.kernel(q, t, l, a)

where the arguments are:

1. **q** is the Command Queue
2. **t** is the Global size as a tuple:
   (x,), (x,y), or (x,y,z)
3. **l** is the Local size as a tuple or None
4. **a** is the list of arguments to pass to the kernel
   - Scalars must be type cast to numpy types; i.e. numpy.uint32(var), numpy.float32(var)

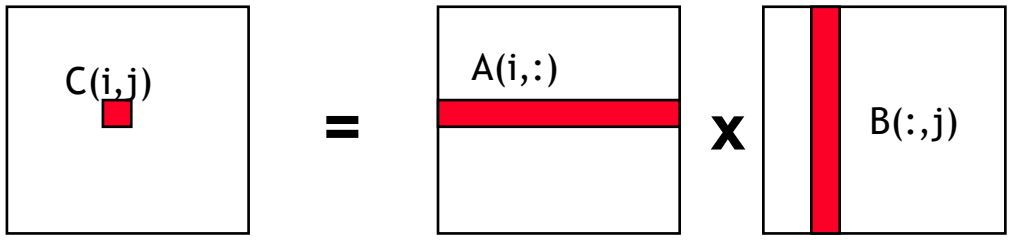
Part 3

OPENCL KERNEL PROGRAMMING

Matrix multiplication: sequential code

We calculate C=AB, where all three matrices are NxN

```
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*N+j] += A[i*N+k] * B[k*N+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).

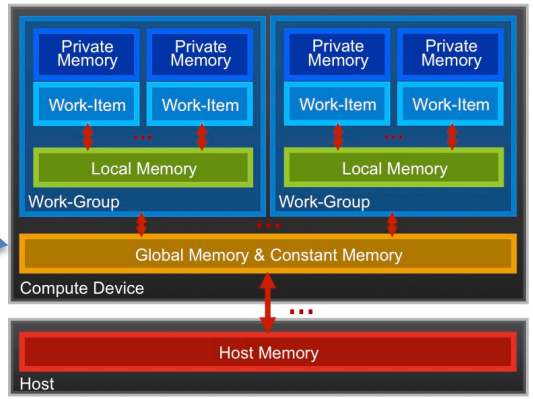
Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A

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

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);
    for (k = 0; k < N; k++) {
      C[i*N+j] += A[i*N+k] * B[k*N+j];
    }
}
```

Use of global memory only in the device

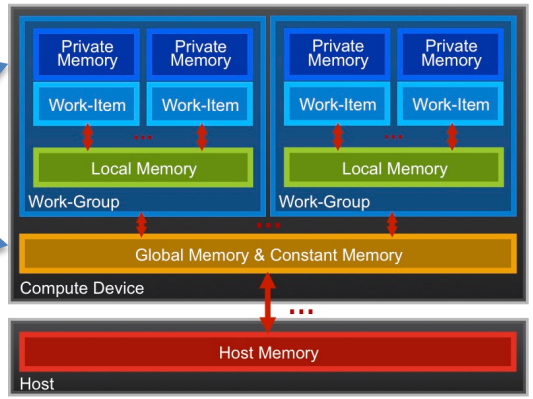


Matrix multiplication: OpenCL kernel

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

use a private scalar **tmp** for intermediate C element values

Use of global and private memory



Matrix multiplication performance

- Matrices are stored in global memory.

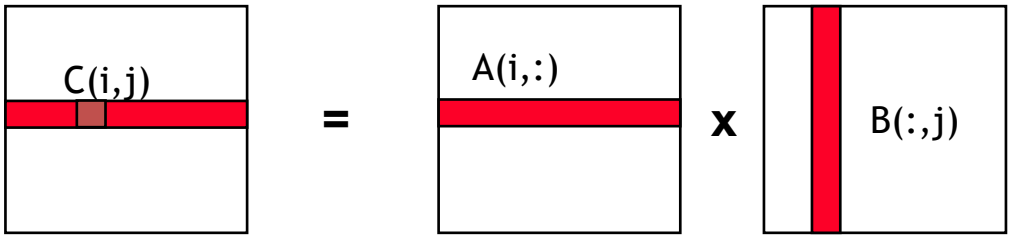
Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9

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

Clear improvement… but could be better

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

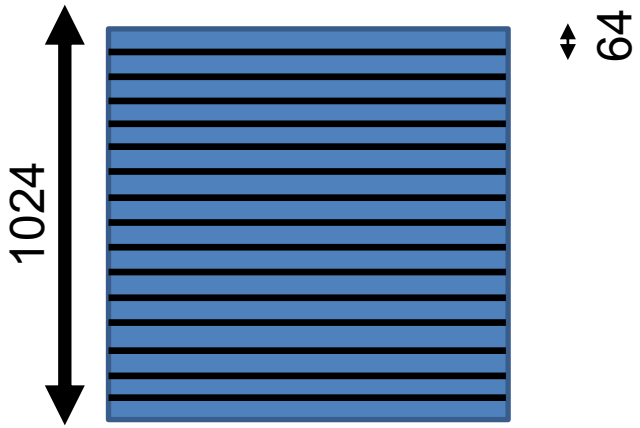
- And with an eye towards future optimizations, let's collect work-items into work-groups with 64 work-items per work-group

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



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 = 0.0f;
   for (k = 0; k < N; k++)
     tmp += A[i*N+k]*B[k*N+j];
   C[i*N+j] = tmp;
  }
}
```

Matrix multiplication performance

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

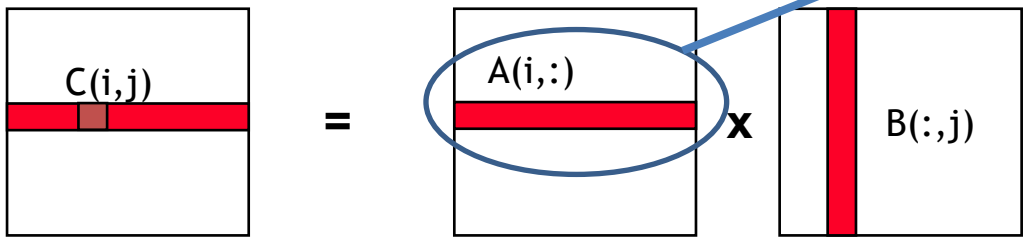
CPU Device is Intel® Xeon® CPU, E5649 @ 2.53GHz
GPU Device is NVIDIA® Tesla® M2090 GPU with a max of 16 compute units, 512 PEs

Mitigated effect… but good start for improvements

Optimizing matrix multiplication

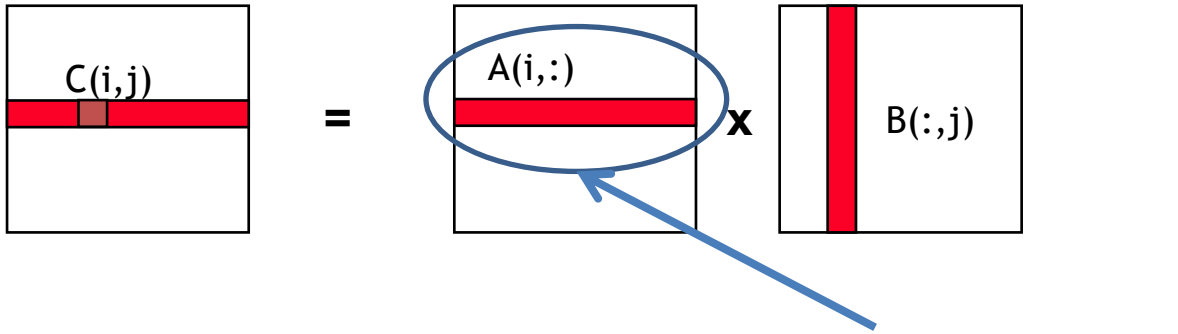
- Notice that, in one row of C, each element reuses the same row of A.

```
int i = get_global_id(0);
…
for (k = 0; k < N; k++)
      tmp += A[i*N+k]*B[k*N+j];
C[i*N+j] = tmp;
…
```



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)

```
__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];   //in private memory

 //copy a row of A into 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];  //only access to global memory for B
     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)

Matrix multiplication performance

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

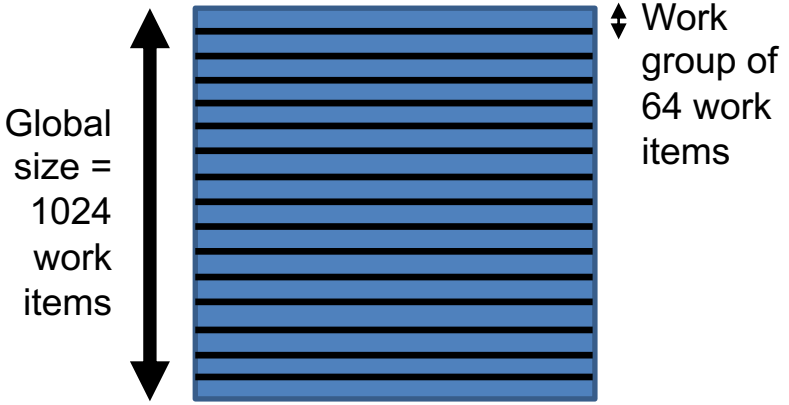
CPU Device is Intel® Xeon® CPU, E5649 @ 2.53GHz
GPU Device is NVIDIA® Tesla® M2090 GPU with a max of 16 compute units, 512 PEs

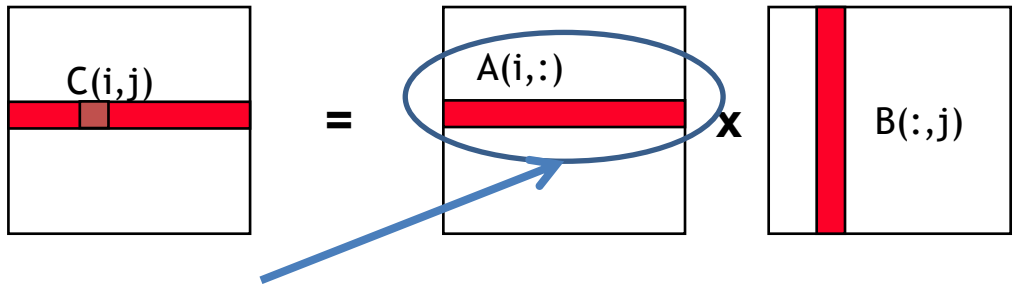
BIG IMPACT ON THE GPU!

Optimizing matrix multiplication

- Each work-item in a work-group also uses the same columns of B

```
int i = get_global_id(0);
…
 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;
 }
…
```



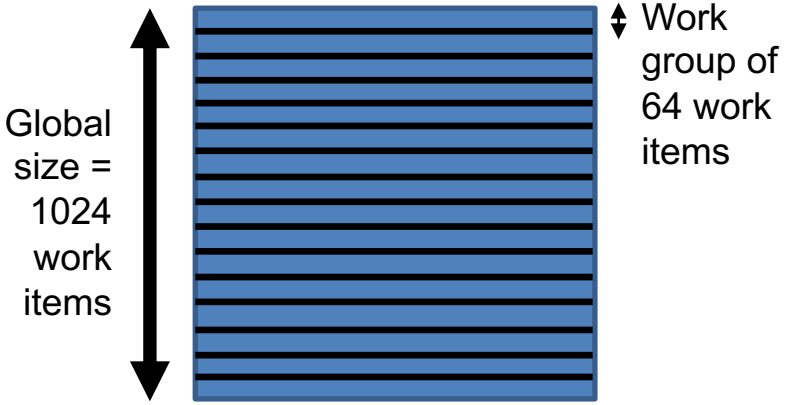


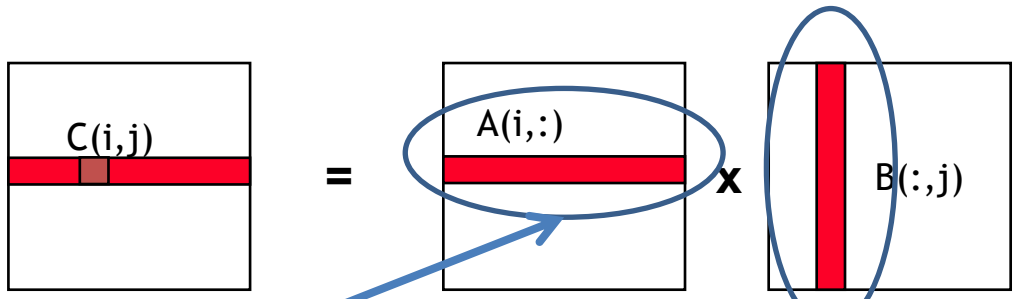
Private memory of each work-item (as in the previous example)

Optimizing matrix multiplication

- Each work-item in a work-group also uses the same columns of B
- Copy these columns in local memory (shared by a work group)

```
int i = get_global_id(0);
…
 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;
 }
…
```





Private memory of each work-item (as in the previous example)

Local memory for each 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 Bwrk 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 performance

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

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

MASSIVE IMPACT ON THE CPU!

Making matrix multiplication *really* fast

We've ignored so far many well-known techniques for making matrix multiplication fast:

- The number of work items must be a multiple of the fundamental machine "vector width" (wavefront on AMD, warp on NVIDIA, 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
- Automatized or empirical tuning

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
Block oriented approach using local	1,534.0	230,416.7

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

HUGE IMPACT ON THE GPU!

Part 4

DIFFERENCES BETWEEN CUDA AND OPENCL

Switching between CUDA and OpenCL is *mainly*:

- Changing the host code syntax
- Changing indexing and naming conventions in the kernel code

… not that hard actually!

Memory Hierarchy Terminology

CUDA

OpenCL

Local – within a thread

Private – within a work-item

Shared – shared between threads in a thread block

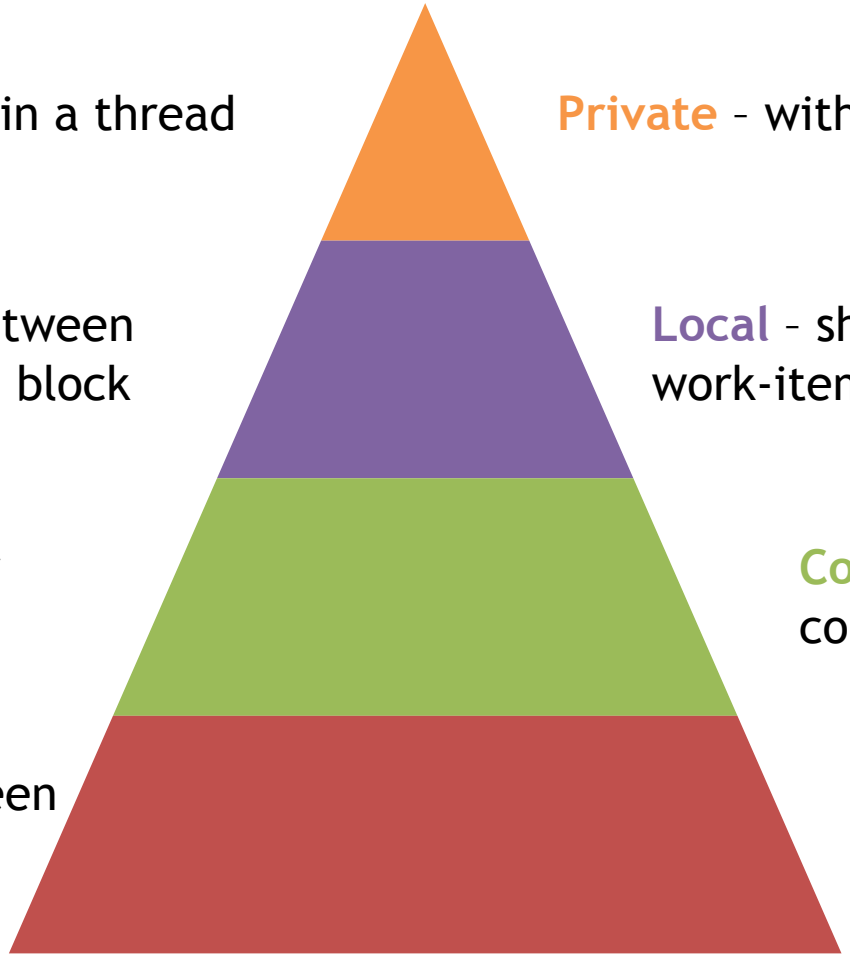
Local – shared between work-items in a work-group

Constant – a cache for constant memory

Constant – a cache for constant memory

Device – shared between all thread blocks

Global – shared between all work-groups



Allocating and copying memory

	CUDA C	OpenCL C

Allocate	`float* d_x;`
	`cudaMalloc(&d_x,`
	`sizeof(float)*size);`
		`cl_mem d_x =`
		`clCreateBuffer(context,`
		`CL_MEM_READ_WRITE,`
		`sizeof(float)*size,`
		`NULL, NULL);`

Host to Device	`cudaMemcpy(d_x, h_x,`
	`sizeof(float)*size,`
	`cudaMemcpyHostToDevice);`
		`clEnqueueWriteBuffer(queue, d_x,`
		`CL_TRUE, 0,`
		`sizeof(float)*size,`
		`h_x, 0, NULL, NULL);`

Device to Host	`cudaMemcpy(h_x, d_x,`
	`sizeof(float)*size,`
	`cudaMemcpyDeviceToHost);`
		`clEnqueueReadBuffer(queue, d_x,`
		`CL_TRUE, 0,`
		`sizeof(float)*size,`
		`h_x, 0, NULL, NULL);`

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. Specify the size by setting the kernel argument

```
clSetKernelArg(kernel, 0,
  sizeof(int)*num_elements,
  NULL);
```

Enqueue a kernel (C)

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};

clEnqueueNDRangeKernel(
        queue, &kernel,
        2, 0, &global, &local,
        0, NULL, NULL);
```

Indexing work

CUDA	OpenCL
gridDim	get_num_groups()
blockIdx	get_group_id()
blockDim	get_local_size()
gridDim * blockDim	get_global_size()
threadIdx	get_local_id()
blockIdx * blockdim + threadIdx	get_global_id()

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

SOME CONCLUDING REMARKS

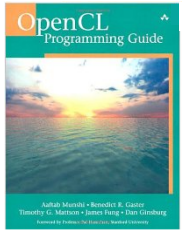
Conclusion

- 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 make 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.

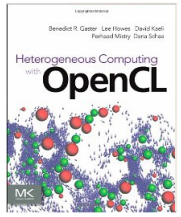
Resources: https://www.khronos.org/opencl/

The OpenCL specification
Surprisingly approachable for a spec!
https://www.khronos.org/registry/cl/

OpenCL reference card
Useful to have on your desk(top)
Available on the same page as the spec.



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