OpenCL C

Matt Sellitto
Dana Schaa
Northeastern University
NUCAR

OpenCL C

- Is used to write kernels when working with OpenCL
- Used to code the part that runs on the device
- Based on C99 with some extensions and restrictions
- Compiled by the OpenCL compiler (with clBuildProgram)

Some Restrictions

- No C standard library (no stdio.h, stdlib.h etc)
- No variable length arrays (declared within a kernel)
- No variable number of arguments in functions
- No recursive function support
- No extern, static, auto, or register keyword support
- No function pointers

Kernels

- Kernels are similar to functions that are called from the host and run on the OpenCL device.
  - They are executed by many instances of parallel work-items (or threads)
  - They have no return type (void)
  - They must be defined with the __kernel qualifier:

```
__kernel void k()
     {
               //kernel code, executed by many parallel threads
     }
```

Non-kernel Functions

- Other functions not declared with the __kernel qualifier are just regular functions
- They can only be called from code running on the device (such as a kernel function or other non-kernel function)

```
// may be called from the host, run with many parallel threads
__kernel void k()
     {
                  f() // each thread calls function f
     }
// just like a normal C function
// can not be called from the host
int f()
       {
                  //do stuff
       }
```

Datatype Support - Scalars

- Work the same way they do in C:

Type in OpenCL Language
bool
char
unsigned char, uchar
short
unsigned short, ushort
int
unsigned int, uint
long
unsigned long, ulong
float
half
size_t
ptrdiff_t
intptr_t
uintptr_t
void

Datatype Support - Vectors

- Work similar to structs with n number fields.
  - Vectors of length 2,3,4,8, and 16 supported
  - Elements accessed similar to C structs (vec.x, vec.y etc)
  - Used for convenience and/or performance

Type	Description
char*n*	A 8-bit signed two's complement integer vector.
uchar*n*	A 8-bit unsigned integer vector.
short*n*	A 16-bit signed two's complement integer vector.
ushort*n*	A 16-bit unsigned integer vector.
int*n*	A 32-bit signed two's complement integer vector.
uint*n*	A 32-bit unsigned integer vector.
long*n*	A 64-bit signed two's complement integer vector.
ulong*n*	A 64-bit unsigned integer vector.
float*n*	A float vector.

  - (will talk more about later)

Datatype Support - Others

- Supports special image types (2d and 3d)
  - (will talk more about later)

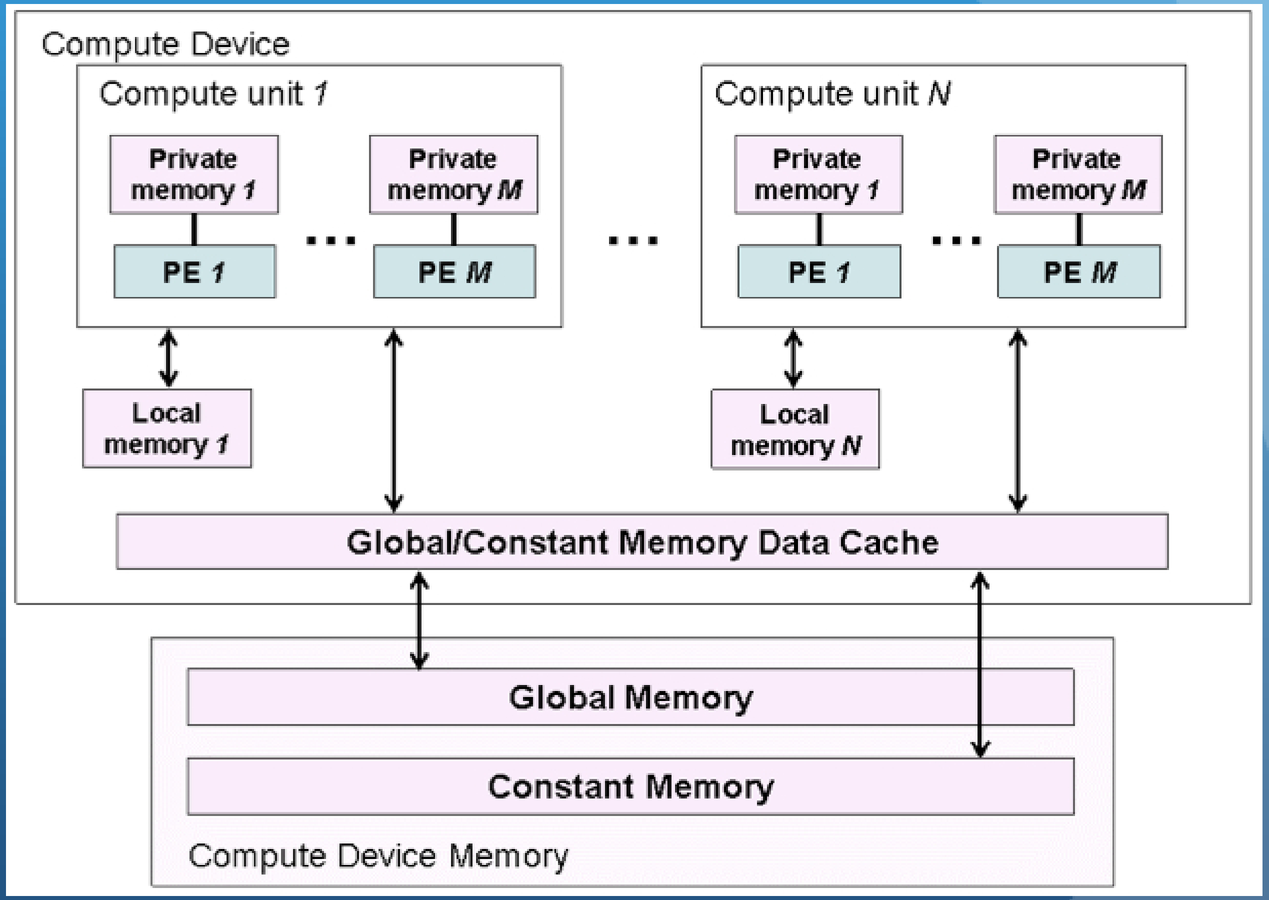
Type	Description
image2d_t	A 2D image. Refer to *section 6.11.13* for a detailed description of this type.
image3d_t	A 3D image. Refer to *section 6.11.13* for a detailed description of this type.
sampler_t	A sampler type. Refer to *section 6.11.13* for a detailed description of this type.
event_t	An event. This can be used to identify async copies from global to local memory and vice-versa. Refer to *section 6.11.10*.

- Also supports fixed length arrays, structs, unions

Address Spaces

- All variables live in 1 of 4 mutually exclusive address spaces:

Memory	Description
Global	Accessible by all work-items
Constant	RO, global
Local	Local to a work-group
Private	Private to a work-item



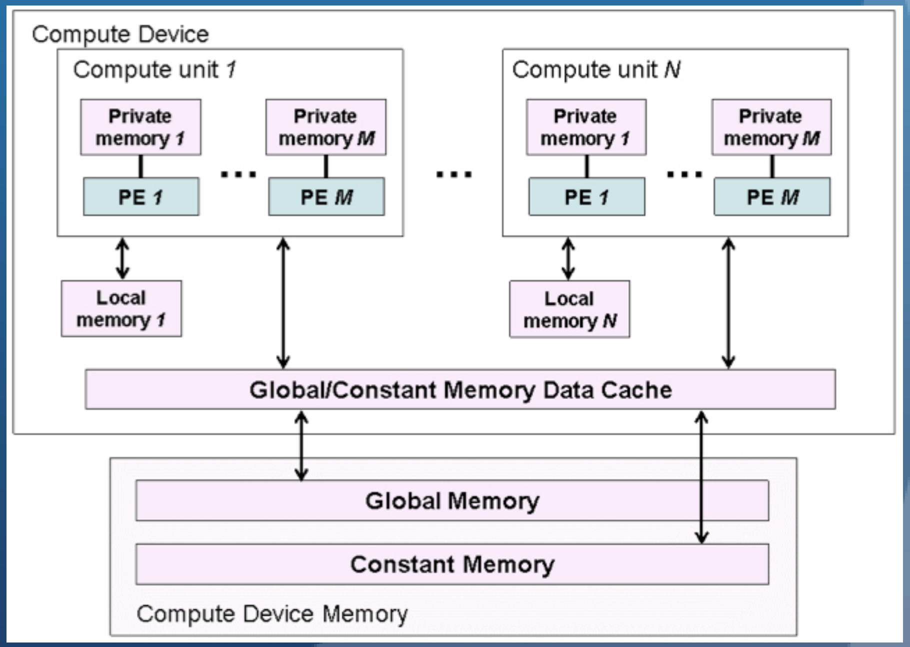
Address Spaces

- Address space qualifiers are used when a variable is declared to specify which region of memory a variable lives in:
  - __private
  - __local
  - __constant
  - __global
- If no qualifier is used the variable defaults to the private address space.

Global Variables

- Visible to all threads within a kernel.
- Space is allocated and initialized before the kernel launch by API calls (clCreateBuffer etc)
- Stored in device main memory
- Images are always implicitly stored in the global address space
- Buffers are usually stored in global memory (but can also be in constant if specified)

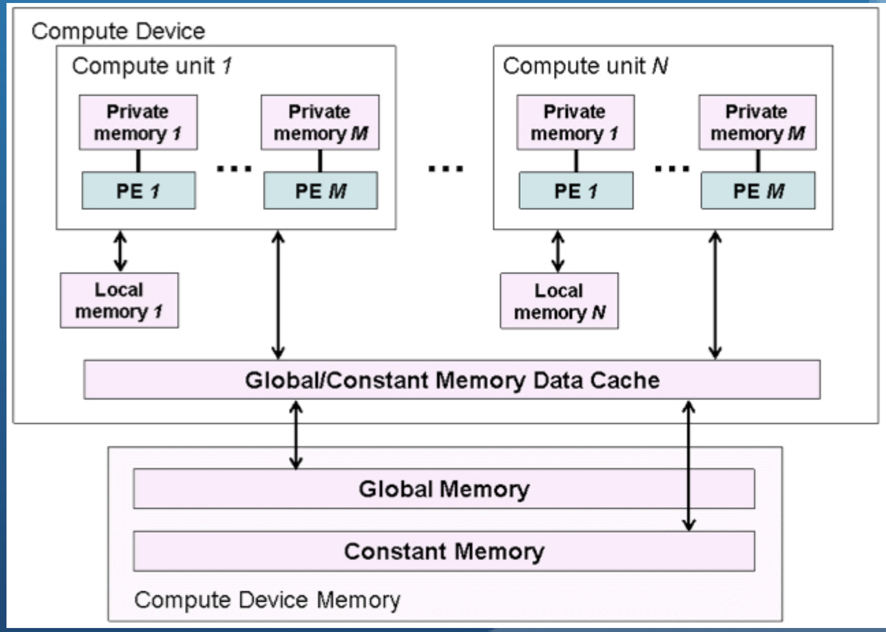
```
/* The array that "x" points to is located
global memory.
Space was allocated with clCreateBuffer() */

__kernel void k(__global int *x)

        {

        }
```



Constant Variables

- Visible to all threads within a kernel.
- Read-only
- Space is allocated and initialized either
  - 1. before the kernel launch by API calls (clCreateBuffer etc).
    - Use CL_MEM_READ_ONLY flag.
  - 2. At program scope
- Stored in device main memory

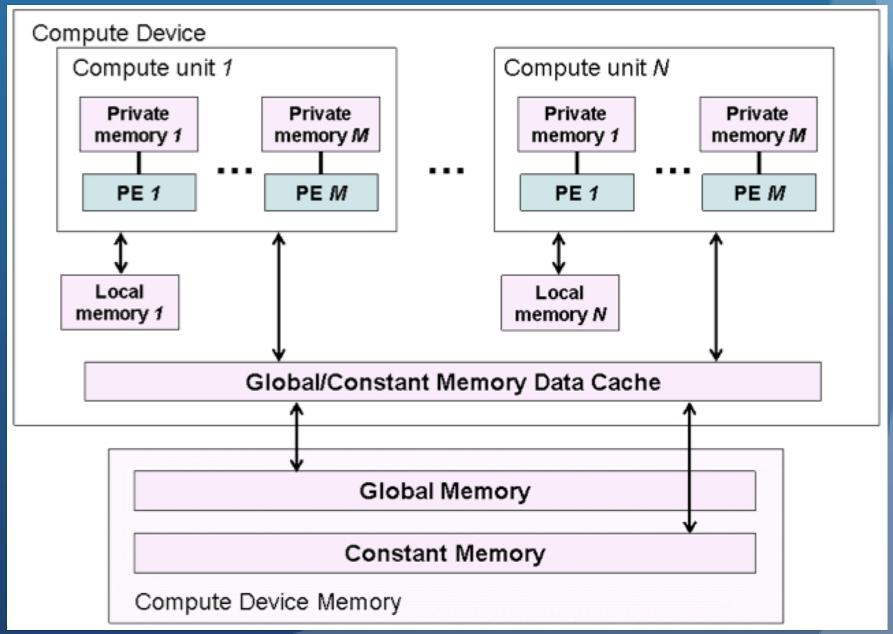
```
__constant C = 4; // program scope, cannot be accessed from host

/* The array that "x" points to is located in constant mem
Space was allocated with clCreateBuffer() */

__kernel void k(__constant int *x)

            {

            }
```



Private Variables

- Generic default address space for variables inside OpenCL code.
- Variables in the private address space are “private” to a work-item (thread)
- Space is allocated automatically
- All kernel and function arguments are in the private address space
- Will usually be stored in registers if possible, may spill into main memory

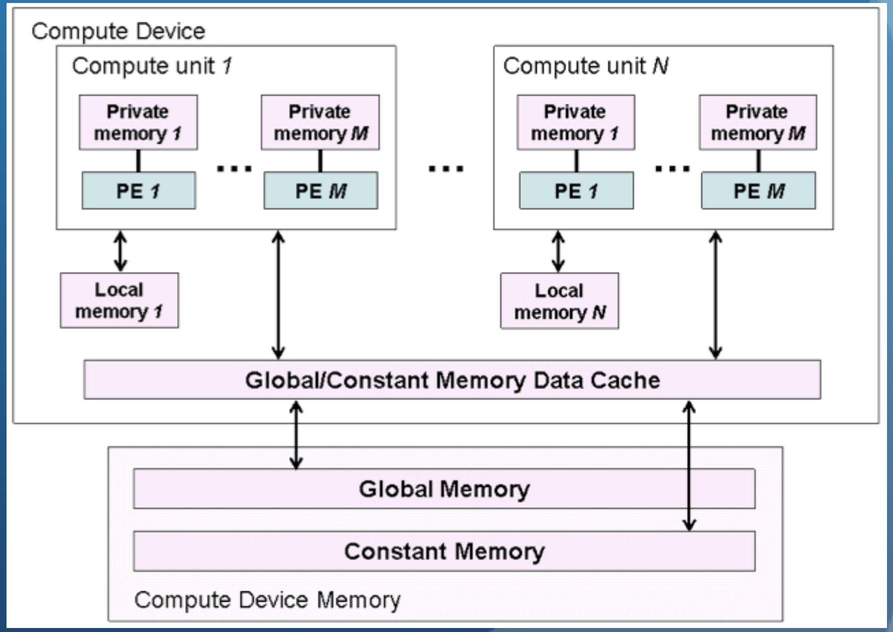
```
/* The array that "x" points to is located in global mem
However, the pointer itself is in private mem */

__kernel void k(__global int *x)

        {

                int p; //in private memory

                int array[4]; //also in private mem

        }
```



Local Variables

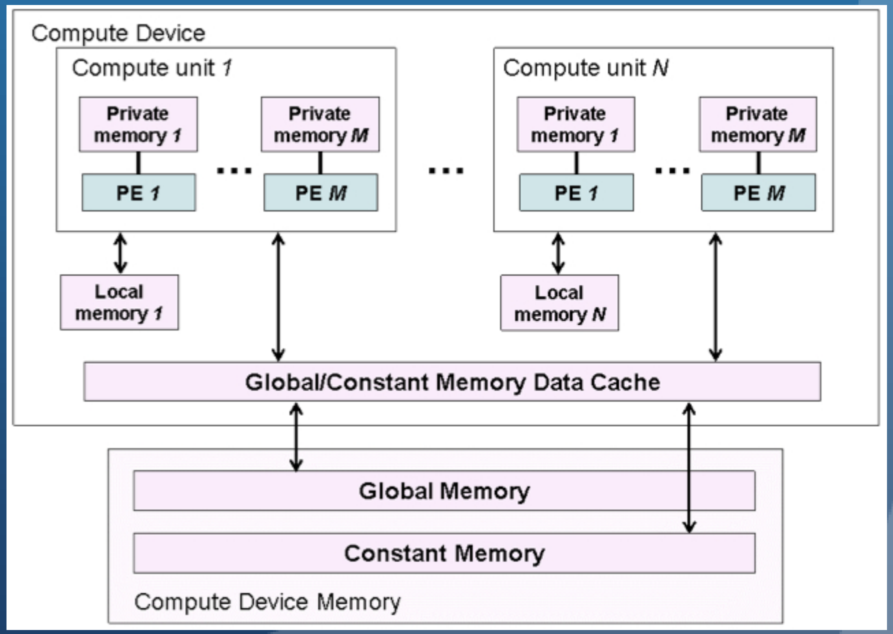
- Visible to all threads within a work-group.
- Space is allocated by either
  - 1. At kernel scope
  - 2. Before kernel launches when passed as an argument
    - Use clSetKernelArg to allocate space()
- Stored in compute-unit memory
- Can not be directly accessed by the host

```
/* The array that "x" points to is located in local mem
Space was allocated with clSetKernelArg() */

__kernel void k(__local int *x)

        {

                __local array[4] // kernel scope local var

        }
```



Memory Size Limitations

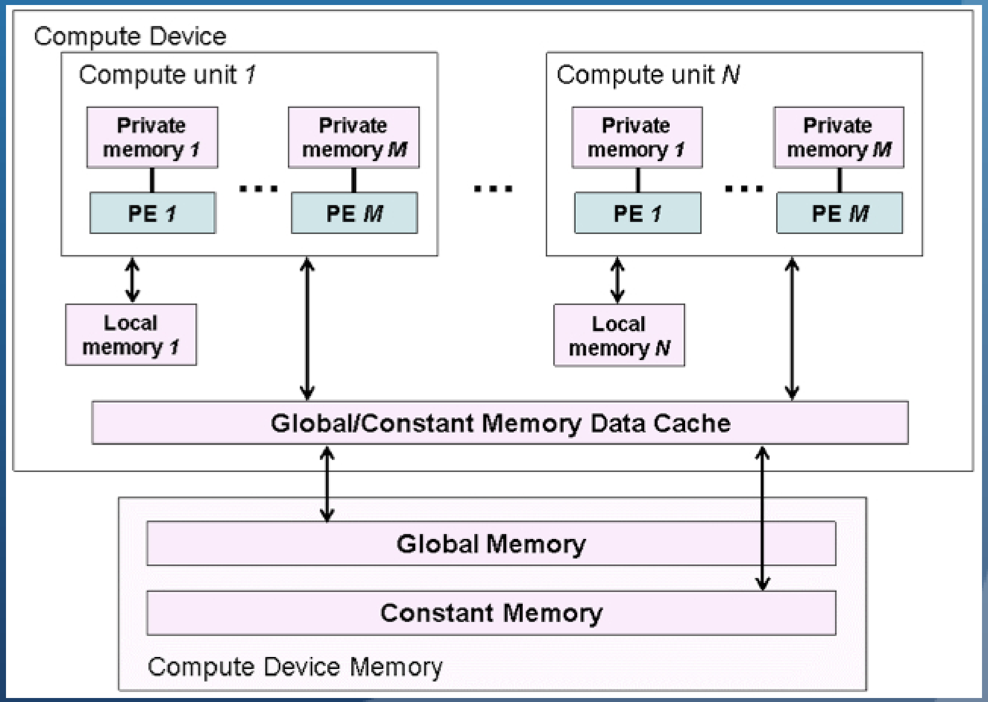
- Private – if too many registers are used per thread, will start to spill into thread-visible main memory
- Global – limited by the amount of main memory of device
- Constant – device limited, usually 64KB per device
- Local – device limited, usually 32KB per compute-unit



Movement between memory spaces

- Movement between memory spaces is explicit
- Simply use the = (assignment) operator.
- Movement between any combinations is fine
  - (except writing to constant memory of course)

```
__kernel void k(__global int *x)
    {
        int p = x[0] // global to private
    }
```



Kernel Arguments

- All arguments passed by pointer to a kernel must be in the __global, __constant, or __local address spaces
- If they are just scalars they can simply be passed via private memory

```
__kernel void k(__global int *x, __constant int *y, __local int *z, int p)
        {
        }
```

Converting Between Types

- Can use regular C casts:

```
int x;
float y = 1.0f;
x = (int) y;
```

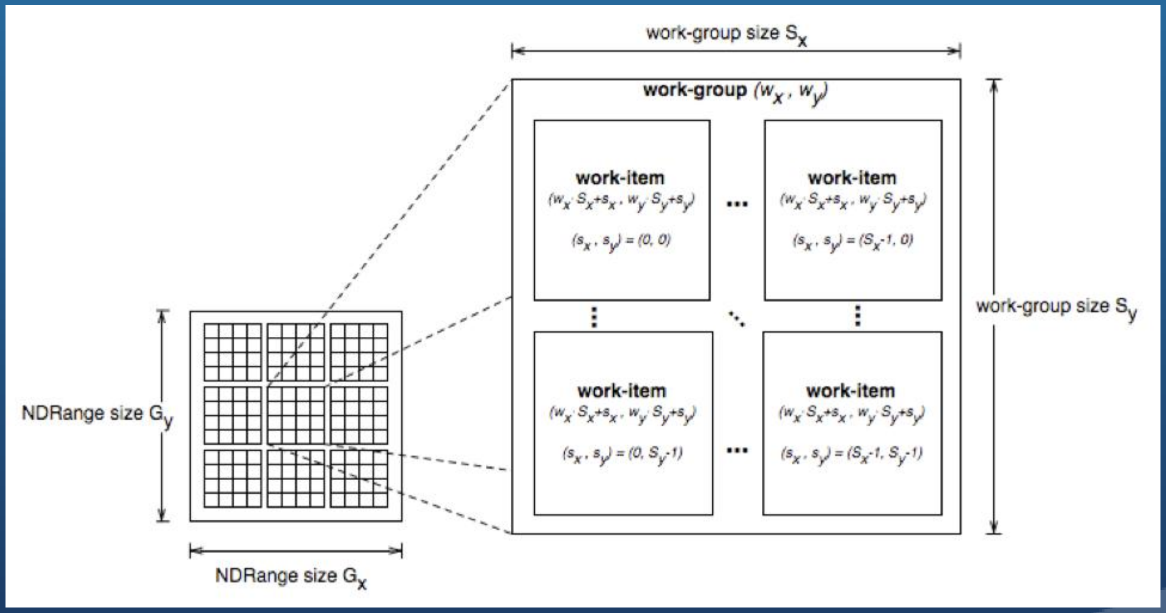
- Safer, more explicit built-in conversion functions:
  - Also supports saturation and rounding modes

```
convert_<destType>(sourceType)

convert_<destType>[_sat][<_rnd>](sourceType) //more general
```

```
// example:
int x = 50000;
char c = convert_char_sat(x)  // c will saturate and become CHAR_MAX
```

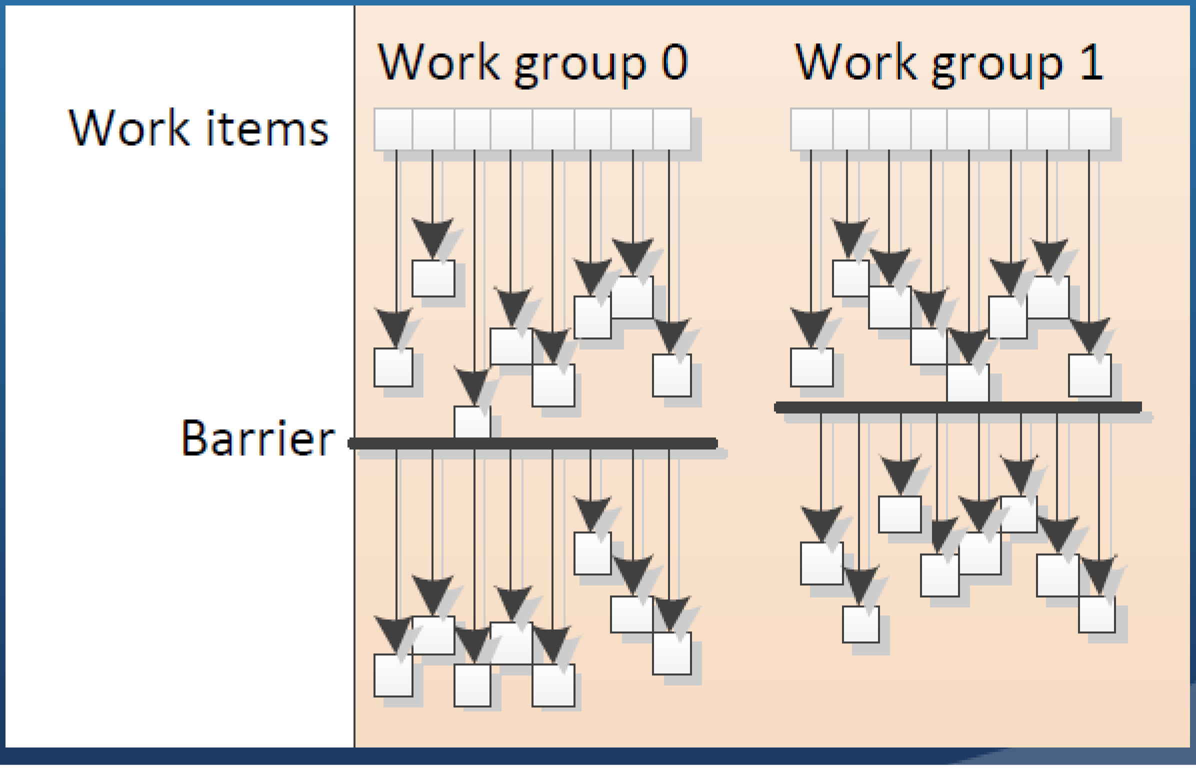
Work-item Functions

Work-item Functions	
Function	**Description**
`get_work_dim()`	Gets number of dimensions
`get_global_size(uint dimIdx)`	Get global work-item size
`get_global_id(uint dimIdx)`	Gets global work-item ID
`get_local_size(uint dimIdx)`	Gets work-group size
`get_local_id(uint dimIdx)`	Gets local work-item ID
`get_num_groups (uint dimIdx)`	Gets number of work-groups
`get_group_id(uint dimIdx)`	Gets work-group ID



Synchronization Function

- barrier(cl_mem_fence_flags *flags*)
- Synchronizes all work-items within a workgroup
- All work-items within a group must reach the barrier() before they continue
- All memory writes before the barrier() from all work-items within the work-group will be visible after the barrier()
  - (for local and/or global memory, depending on flags used)



Other Built-in Functions

- OpenCL C comes with all sorts of built-in functions
  - Math
  - Vectors
  - Comparison
  - Integer
  - Images
  - And more (all documented in the spec)

Preprocessor

- Supports preprocessor macros
  - #define, #include, #ifdef, etc
  - Macros can also be defined when compiling
    - (with clBuildProgram)
  - #pragma OPENCL <xxxx> is used to enable things like extensions (will talk more about that next class)

Matrix Multiplication Exercise

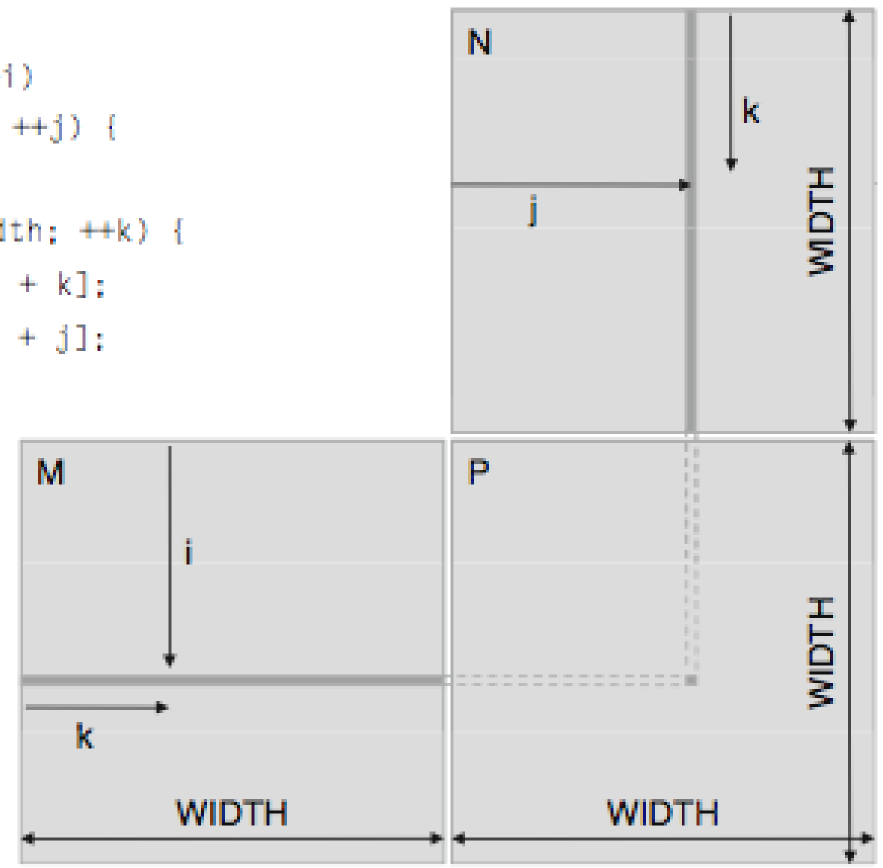
- In the Class4 folder there is a matrixMultiplication project for you to copy to your home directory.
- The goal is to write a parallel matrix multiplication kernel to run on the GPU.
- Almost all the host side code in matrixMultiplication.cpp is complete, you only need to write the kernel "matmul" in matmul.cl (currently blank)
- The kernel takes two int matrices A and B (as single arrays), the matrix width N (matrix is square) and stores the result in matrix C

Matrix Multiplication Exercise

- Some variables in the code you may want to change:
  - N – dimensions of the matrix
  - Weather or not to check result against CPU version (host_check)
- The program also prints out the time it takes to execute
- Optional: Once you get a working matrix multiply kernel, can you improve it? Call this kernel “matmul2” and see if you can make it faster than the first one.

Serial Matrix Multiplication

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
      float sum = 0;
      for (int k  = 0; k < Width; ++k) {
        float a = M[i * width + k];
        float b = N[k * width + j];
        sum += a * b;
      }
      P[i * Width + j] = sum;
    }
}
```

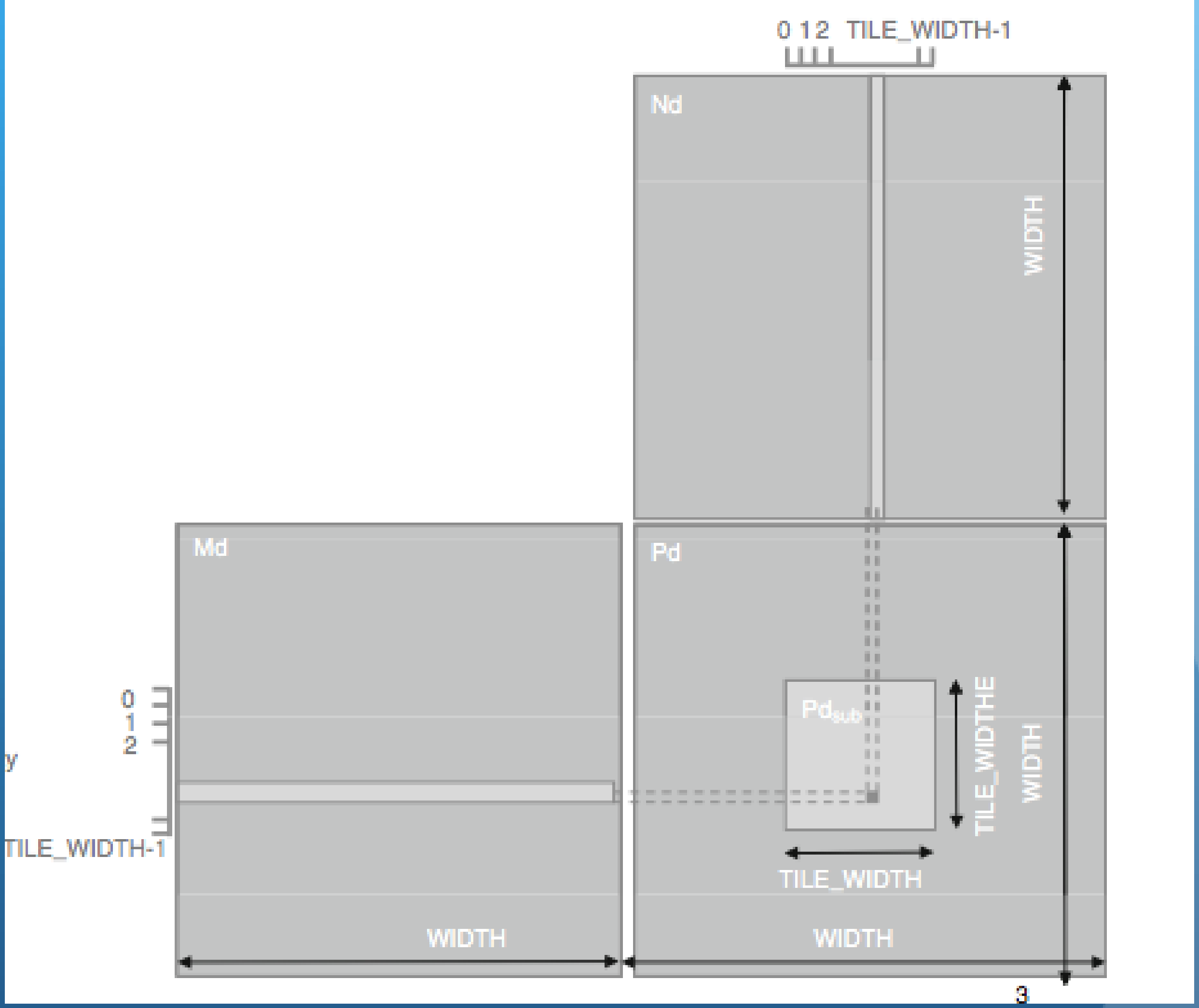


Parallel Matrix Multiply in OpenCL

```
// basic matrix multiply
__kernel void matmul(__global int *A, __global int *B, __global int *C, int N)
{

        int gx = get_global_id(0);
        int gy = get_global_id(1);

        int Cvalue = 0;

        for(int k = 0; k < N; k++)
        {
                Cvalue += A[gy*N+k] * B[k*N+gx];

        }

        C[gy*N+gx] = Cvalue;
}
```

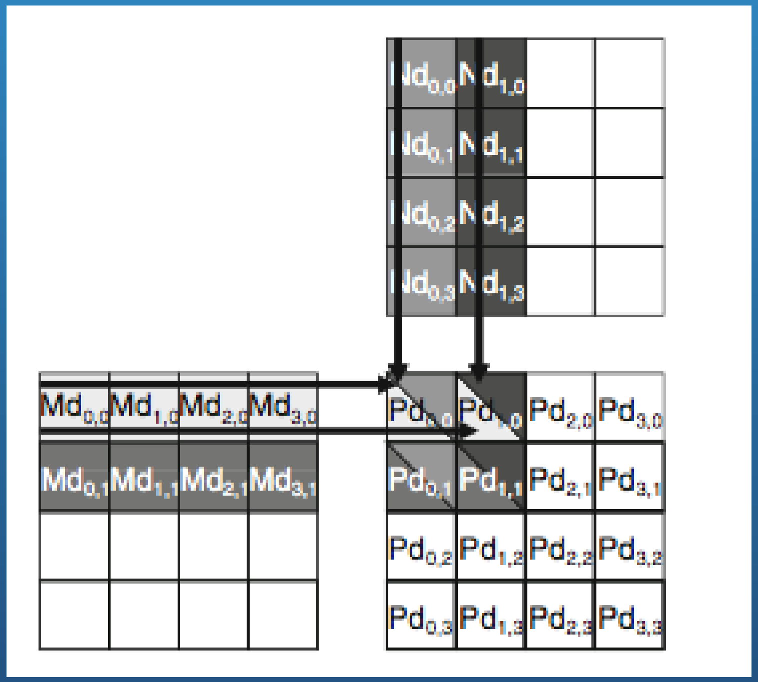
Matrix Multiplication 2
- Tiled Using Local Memory

- Each tile can be made out of a work-group.
- Since the threads in that work-group will use some of the same elements of A and B they can be brought in to local memory to get better memory performance.



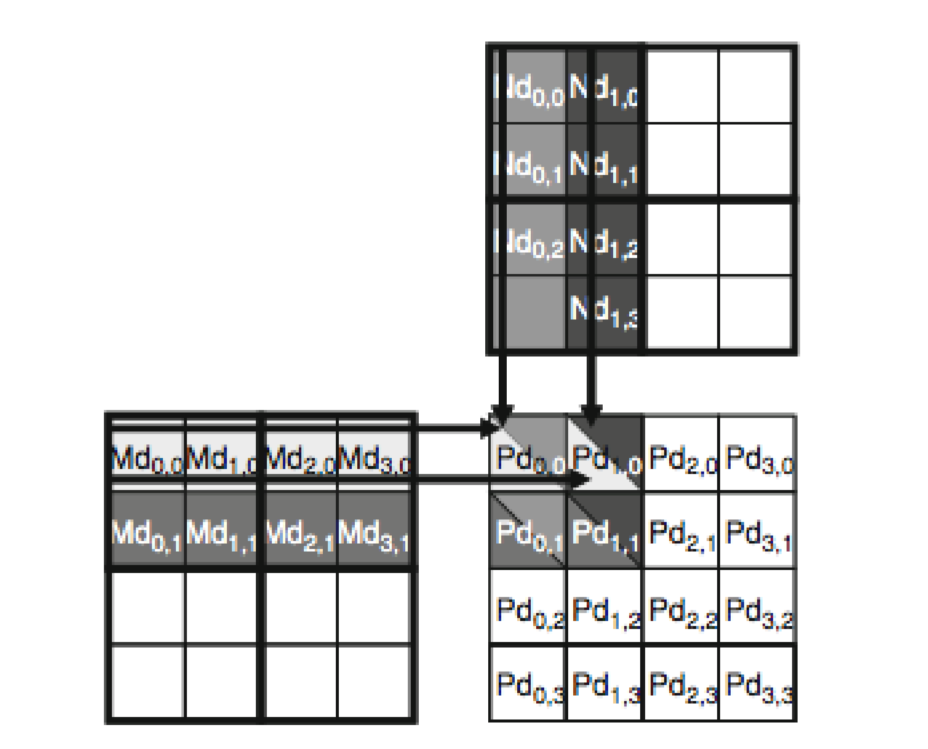
Matrix Multiplication 2
- Tiled Using Local Memory

- Some of the work-items use duplicate elements from A and B as other work-items in their group:



Matrix Multiplication 2
- Tiled Using Local Memory

- Therefore, we can work on one work-group sized tile of A and B at a time by bringing it into local memory and operating on it.



Matrix Multiplication 2
- Tiled Using Local Memory

```
// tiled matrix multiply using local memory
__kernel void matmul2(__global int *A, __global int *B, __global int *C, int N, __local int *l_A,
__local int *l_B)
{

	int gx = get_global_id(0);
	int gy = get_global_id(1);
	int lx = get_local_id(0);
	int ly = get_local_id(1);
	int tile_width = get_local_size(0); // tile width is local size dim

	int Cvalue = 0;

	// loop over the number of tiles that fit in one matrix dimension
	for(int m = 0; m < (N/tile_width); m++)
	{
		// fill in the local data
		l_A[ly*tile_width+lx] = A[gy*N +(m*tile_width+lx)];
		l_B[ly*tile_width+lx] = B[(m*tile_width+ly)*N + gx];
		barrier(CLK_LOCAL_MEM_FENCE);


		// accumulate Cvalue only for the local data
		for(int k = 0; k < get_local_size(0); k++)
		{
			Cvalue += l_A[ly*tile_width + k] * l_B[k*tile_width+lx];
		}
		barrier(CLK_LOCAL_MEM_FENCE);
	}


	C[gy*N+gx] = Cvalue;
}
```

Matrix Multiplication 2
- Tiled Using Local Memory

- Using local memory I was able to get about 35-40% performance improvement when using N = 4096 and work-group size of 16x16

- Questions?