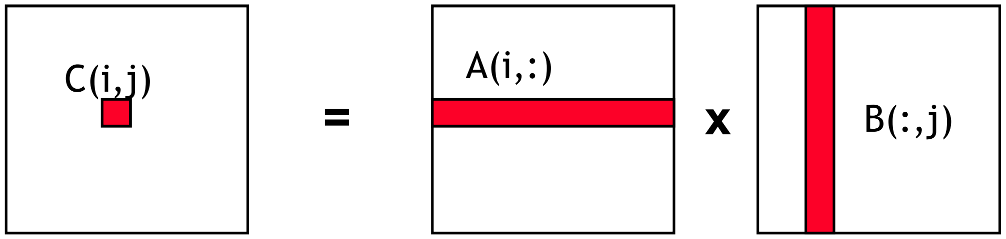
Lecture 6

UNDERSTANDING THE OPENCL MEMORY HIERARCHY

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.

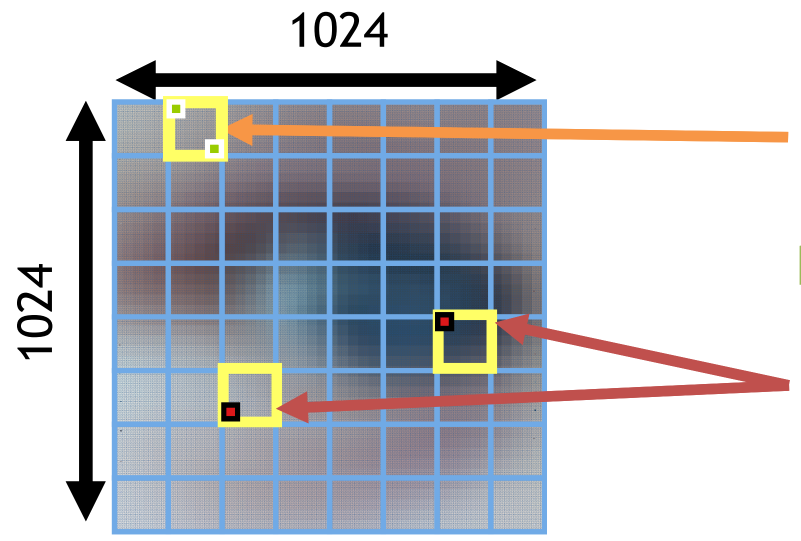


Dot product of a row of A and a column of B for each element of C

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

An N-dimensional domain of work-items

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



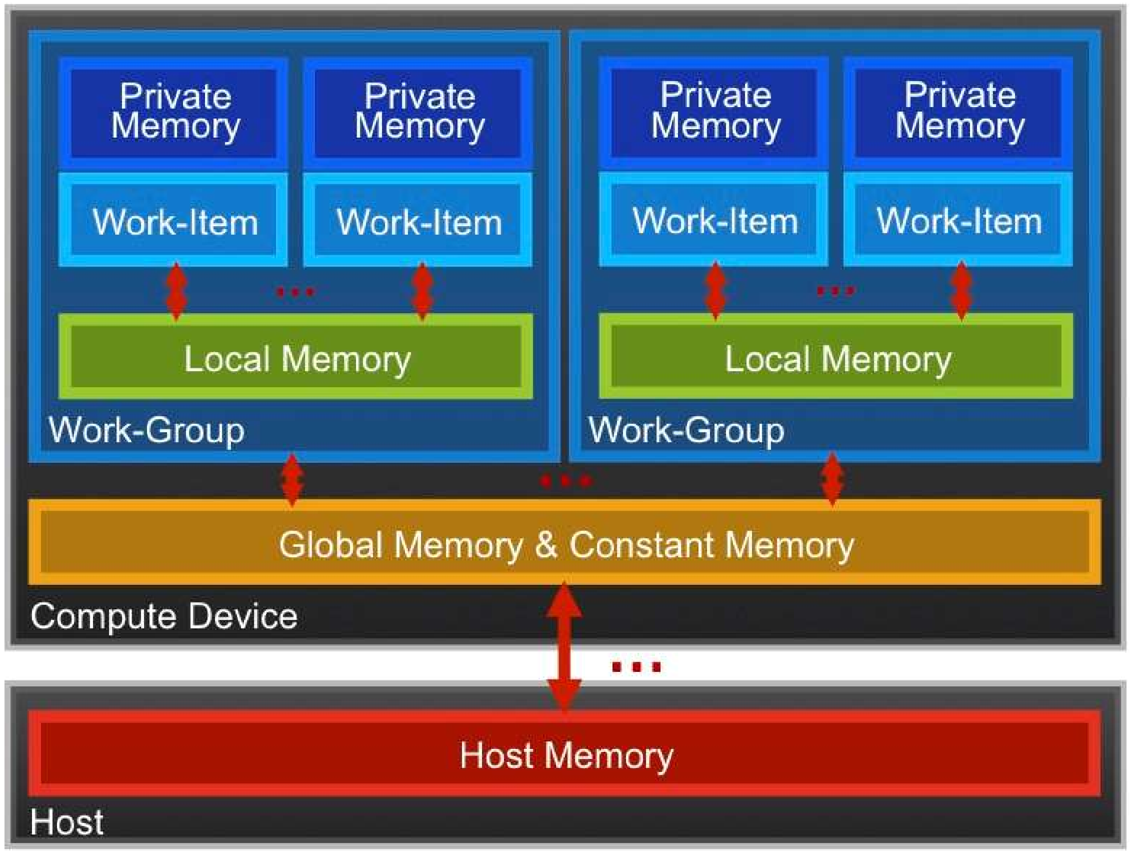
Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

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

OpenCL Memory model

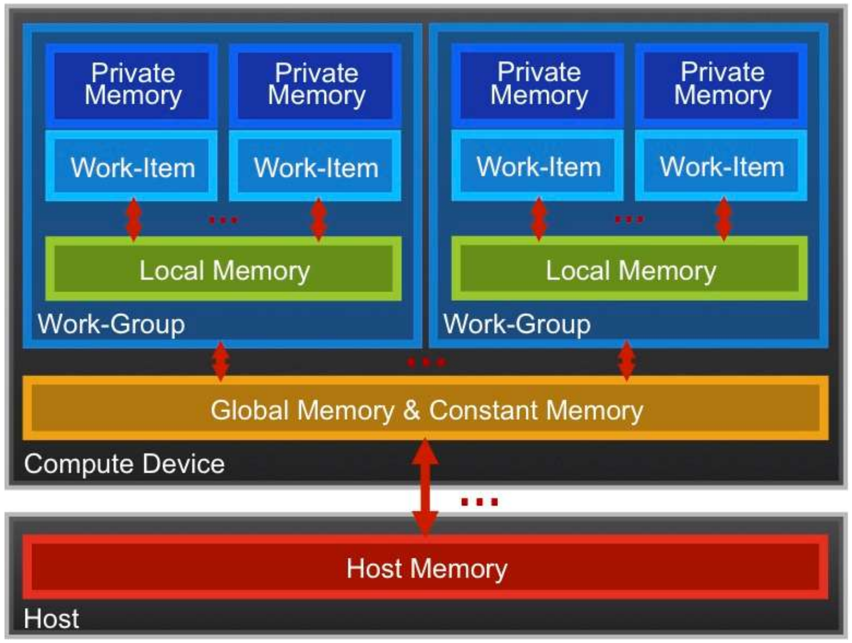
- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU



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

OpenCL Memory model

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



Memory management is **explicit**:
O(1-10) Gbytes/s bandwidth to discrete GPUs for
Host <-> 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*

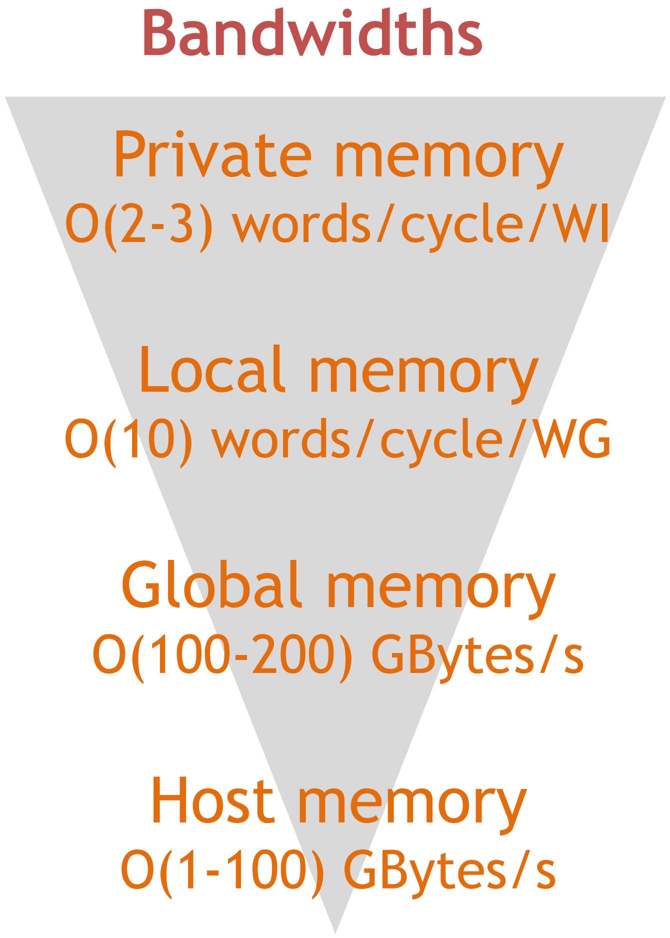
- Tens of KBytes per Compute Unit
  - As multiple Work-Groups will be running on each CU, this means only a fraction of the total Local Memory size is available to each Work-Group
- Assume O(1-10) KBytes of Local Memory per Work-Group
  - Your kernels are responsible for transferring data between Local 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

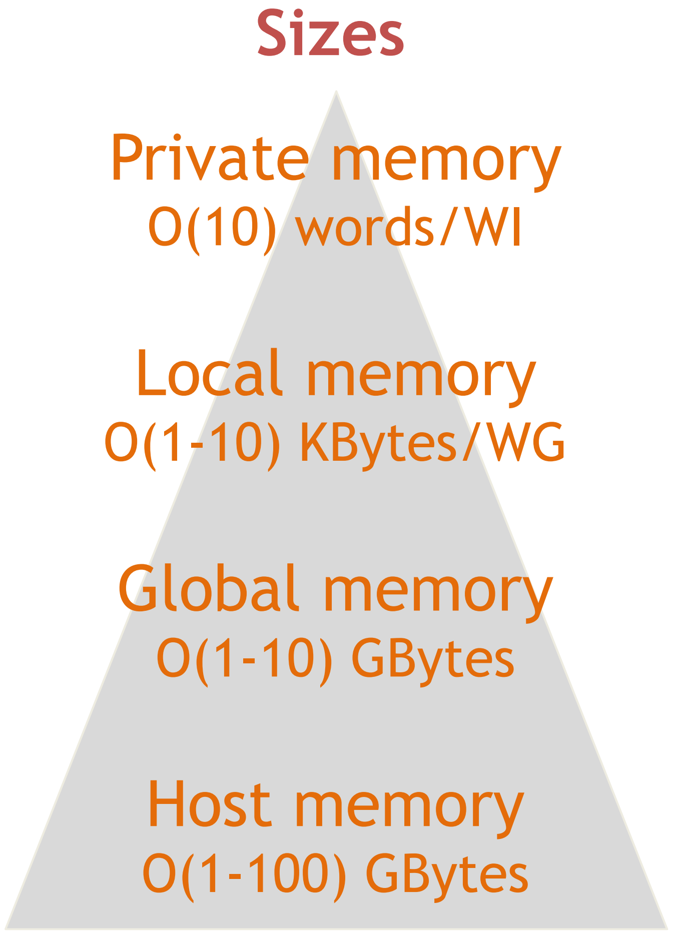
* Typical figures for a 2013 GPU

Local Memory

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

The Memory Hierarchy





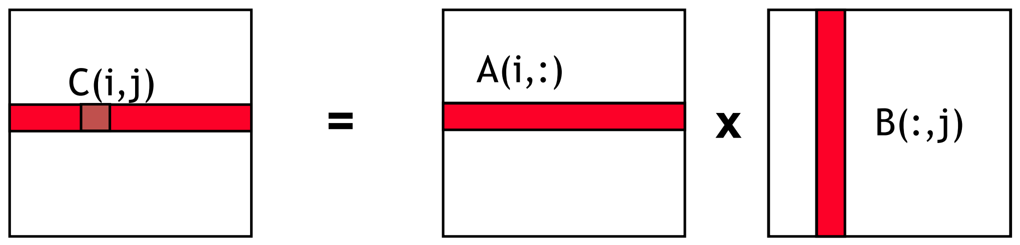
Speeds and feeds approx. for a high-end discrete GPU, circa 2011

Memory Consistency

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

Optimizing matrix multiplication

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

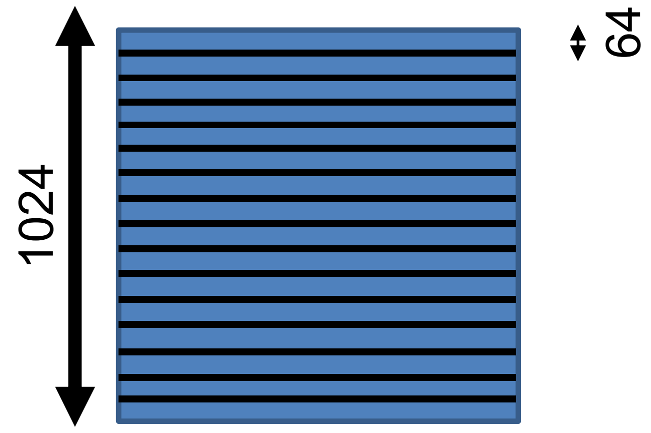


Dot product of a row of A and a column of B for each element of C

- 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



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

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)

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::vector
  int Mdim, N
  int i, err;
  int szA, sz
  double star
  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);

  // 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);
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

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

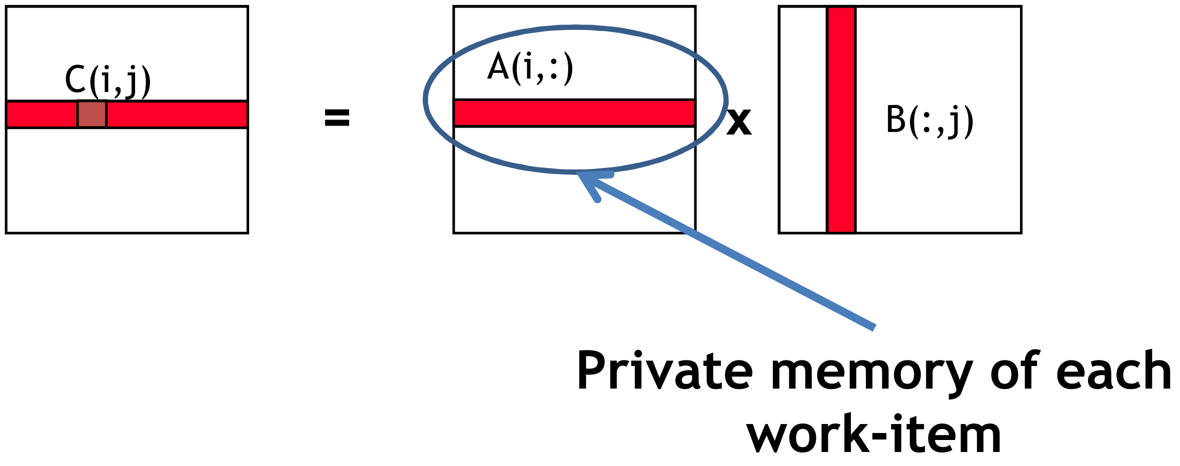
This has started to help.

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

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

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)

Matrix multiplication performance

- Matrices are stored in global memory.

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

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

Big impact!

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

Why using too much private memory can be a good thing

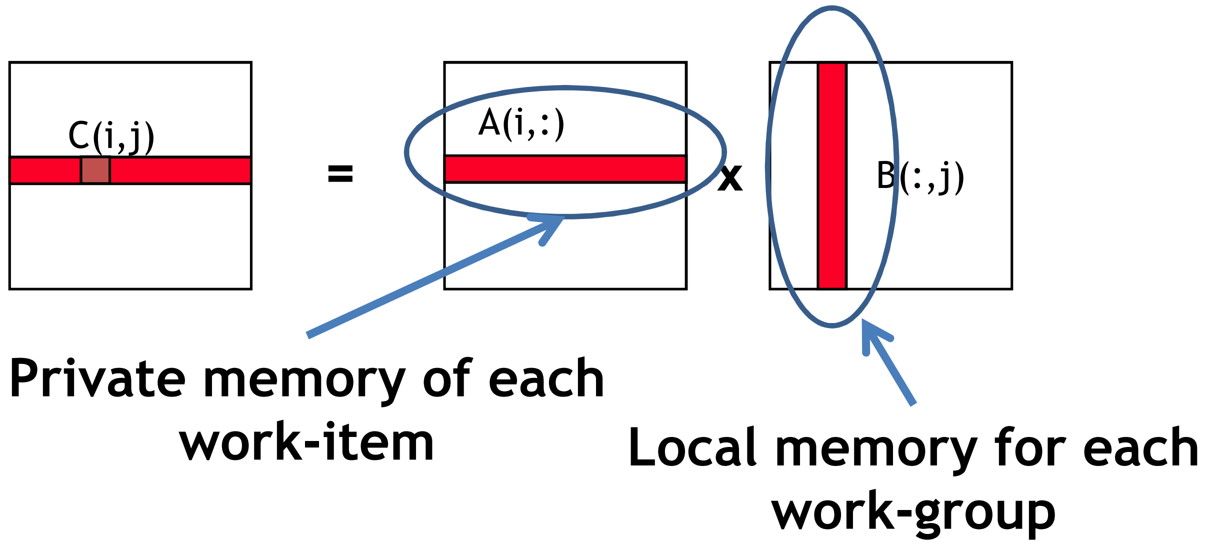
- In reality private memory is just hardware registers, so only dozens of these are available per work-item
- Many kernels will allocate too many variables to private memory
- So the compiler already has to be able to deal with this
- It does so by *spilling* excess private variables to (global) memory
- You still told the compiler something useful – that the data will only be accessed by a single work-item
- This lets the compiler allocate the data in such as way as to enable more efficient memory access

Exercise 7: using private memory

- Goal:
  - Use private memory to minimize memory movement costs and optimize performance of your matrix multiplication program
- Procedure:
  - Start with your matrix multiplication solution
  - Modify the kernel so that each work-item copies its own row of A into private memory
  - Optimize step by step, saving the intermediate versions and tracking performance improvements
- Expected output:
  - A message to standard output verifying that the matrix multiplication program is generating the correct results
  - Report the runtime and the MFLOPS

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)


C(i,j)
=
A(i,:)
x
B(:,j)
Private memory of each work-item
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 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)

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

```
i                                                                   ,
{

                                                                    nd(), true);
                                                                    nd(), true);

                                                                    ONLY,
                                                                     * 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);
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

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

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

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

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
  - We modified the matrix multiplication program provided with the NVIDIA OpenCL SDK to work with our test suite to produce the blocked results on the following slide. This used register blocking with block sizes mapped onto the GPU's warp size

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

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

Biggest impact so far!

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

Exercise 8: using local memory

- Goal:
  - Use local memory to minimize memory movement costs and optimize performance of your matrix multiplication program
- Procedure:
  - Start with your matrix multiplication solution that already uses private memory from Exercise 7
  - Modify the kernel so that each work-group collaboratively copies its own column of B into local memory
  - Optimize step by step, saving the intermediate versions and tracking performance improvements
- Expected output:
  - A message to standard output verifying that the matrix multiplication program is generating the correct results
  - Report the runtime and the MFLOPS
- Extra:
  - Look at the fast, blocked implementation from the NVIDIA OpenCL SDK example. Try running it and compare to yours