OpenCL

Lecture 3

Optimising OpenCL performance

Based on material by Benedict Gaster and Lee Howes (AMD), Tim Mattson (Intel) and several others.

University of BRISTOL

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **OpenCL overview**
- **Exploring the spec through a series of examples**
  - **Vector addition:**
    - *the basic platform layer*
  - **Matrix multiplication:**
    - *writing simple kernels*
  - **Optimizing matrix multiplication:**
    - *work groups and the memory model*
- **A survey of OpenCL 1.1**

Linear algebra

- **Definition:**
  - The branch of mathematics concerned with the study of vectors, vector spaces, linear transformations and systems of linear equations.
- **Example: Consider the following system of linear equations**

$$
\begin{aligned}
x + 2y + z &= 1 \\
x + 3y + 3z &= 2 \\
x + y + 4z &= 6
\end{aligned}
$$

  - This system can be represented in terms of vectors and a matrix as the classic "Ax = b" problem.

$$
\begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & 3 \\ 1 & 1 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 6 \end{pmatrix}
$$

Solving Ax=b

- **LU Decomposition:**
  - transform a matrix into the product of a lower triangular and upper triangular matrix. It is used to solve a linear system of equations.

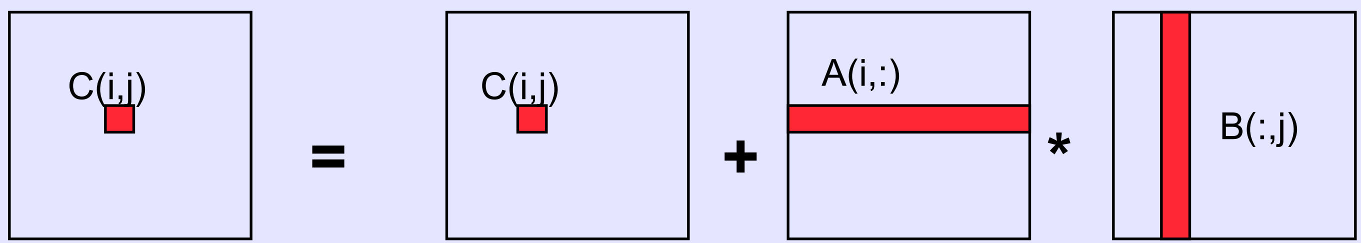
$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & 3 \\ 1 & 2 & 4 \end{pmatrix}$$

$$L \cdot U = A$$

- Solving for x — Given a problem Ax=b
  - Ax=b — LUx=b
  - $Ux=(L^{-1})b$ — $x = (U^{-1})L^{-1}b$

Matrix multiplication: sequential code

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



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

Matrix multiplication performance

- **Results on an Apple laptop with an Intel CPU.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167

Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Matrix multiplication: OpenCL kernel (1/4)

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

Matrix multiplication: OpenCL kernel (2/4)

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

```
__kernel void mat_mul(
  const int Mdim, const int Ndim, const int Pdim,
  __global float *A, __global float *B, __global float *C)
```

Mark as a kernel function and specify memory qualifiers

Matrix multiplication: OpenCL kernel (3/4)

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

```
i = get_global_id(0);
j = get_global_id(1);
```

Remove outer loops and set work-item coordinates

Matrix multiplication: OpenCL kernel (4/4)

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

Matrix multiplication: OpenCL kernel

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

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

Matrix multiplication host program

```
#include "mult.h"
int main(int argc, char **argv)
{
   float *A, *B, *C;
   int Mdim, Ndim, Pdim;
   int err, szA, szB, szC;
   size_t global[DIM];
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
   cl_command_queue commands;
   cl_program program;
   cl_kernel kernel;
   cl_uint nd;
   cl_mem a_in, b_in, c_out;
   Ndim = ORDER;
   Pdim = ORDER;
   Mdim = ORDER;
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);

   err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
   context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
   commands = clCreateCommandQueue(context, device_id, 0, &err);

   a_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szA, NULL, NULL);
   b_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szB, NULL, NULL);
   c_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * szC, NULL, NULL);

   err = clEnqueueWriteBuffer(commands, a_in, CL_TRUE, 0, sizeof(float) * szA, A, 0, NULL, NULL);
   err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);

   *program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource,  NULL, &err);
   err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

   *kernel = clCreateKernel(*program, "mmul", &err);
   err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
   err |= clSetKernelArg(*kernel, 1, sizeof(int), &Ndim);
   err |= clSetKernelArg(*kernel, 2, sizeof(int), &Pdim);
   err |= clSetKernelArg(*kernel, 3, sizeof(cl_mem), &a_in);
   err |= clSetKernelArg(*kernel, 4, sizeof(cl_mem), &b_in);
   err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);

   global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *ndim = 2;
   err = clEnqueueNDRangeKernel(commands, kernel, ndim, NULL, global, NULL, 0, NULL, NULL);
   clFinish(commands);
   err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
   test_results(A, B, c_out);
}
```

Matrix multiplication host program

```
#include "mult.h"
int main(int argc, char **argv)
{
   float *A, *B, *C;
   int Mdim, Ndim, Pdim;
   int err, szA, szB, szC;
   size_t global[DIM];
   size_t lo
   cl_devic
   cl_conte
   cl_comm
   cl_progr
   cl_kerne
   cl_uint nd;
   cl_mem a_in, b_in, c_out;
   Ndim = ORDER;
   Pdim = ORDER;
   Mdim = ORDER;
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
   err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
   context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
   commands = clCreateCommandQueue(context, device_id, 0, &err);

   a_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szA, NULL, NULL);
   b_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szB, NULL, NULL);
   c_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * szC, NULL, NULL);

   err = clEnqueueWriteBuffer(commands, a_in, CL_TRUE, 0, sizeof(float) * szA, A, 0, NULL, NULL);

   &err);

   err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
   err |= clSetKernelArg(*kernel, 1, sizeof(int), &Ndim);
   err |= clSetKernelArg(*kernel, 2, sizeof(int), &Pdim);
   err |= clSetKernelArg(*kernel, 3, sizeof(cl_mem), &a_in);
   err |= clSetKernelArg(*kernel, 4, sizeof(cl_mem), &b_in);
   err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);

   global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *ndim = 2;
   err = clEnqueueNDRangeKernel(commands, kernel, ndim, NULL, global, NULL, 0, NULL, NULL);
   clFinish(commands);
   err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
   test_results(A, B, c_out);
}
```

Note: This isn't as bad as you first think.
This is almost the same as the host code we wrote for vector add.
It's "boilerplate" … you get it right once and just re-use it.

Matrix multiplication host program

```
#include "mult.h"
int main(int argc, char **argv)
{
   float *A, *B, *C;
   int Mdim, Ndim, Pdim;
   int err, szA, szB, szC;
   size_t global[DIM];
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
   cl_command_queue commands;
   cl_pro
   cl_ker
   cl_uin
   cl_me
   Ndim = ORDER;
   Pdim = ORDER;
   Mdim = ORDER;
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

Declare and initialize data

```
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

Setup the platform

```
a_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szA, NULL, NULL);
b_in  = clCreateBuffer(context, CL_MEM_R
c_out = clCreateBuffer(context, CL_MEM_W

err = clEnqueueWriteBuffer(commands, a_
err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);
```

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

```
*program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource,  NULL, &err);
err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

*kernel = clCreateKernel(*program, "mmul", &err);
err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
err |= clSetKernelArg(*kernel, 1, sizeof(int), &Ndim);
err |= clSetKernelArg(*kernel, 2, sizeof(int), &Pdim);
err |= clSetKernelArg(*kernel, 3, sizeof(cl_mem), &a_in);
err |= clSetKernelArg(*kernel, 4, sizeof(cl_mem), &b_in);
err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);
```

Build the program, define the kernel and setup arguments

```
global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *ndim = 2;
err = clEnqueueNDRangeKernel(commands, ker
clFinish(commands);
err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
}
```

Run the kernel and collect results

Matrix multiplication host program

```
#include
int main(
{
   float *A
   int Mdi
   int err,
   size_t
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
```

```
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

The only parts new to us …

1. 2D ND Range set to dimensions of C matrix
2. Local sizes set to NULL in clEnqueueNDRangeKernel() to tell system to pick local dimensions (work-group size) for us.

```
err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);
```

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

```
global[0] = (size_t) Ndim;      global[1] = (size_t) Mdim;      *ndim = 2;
err = clEnqueueNDRangeKernel(commands, kernel, ndim, NULL, global, NULL, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
```

```
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);
```

```
global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *ndim = 2;
err = clEnqueueNDRangeKernel(commands, kernel, ndim, NULL, global, NULL, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
}
```

Matrix multiplication performance

- Results on an Apple laptop with an NVIDIA GPU and an Intel CPU. Matrices are stored in global memory.

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
CPU: C(i,j) per work item, all global	744

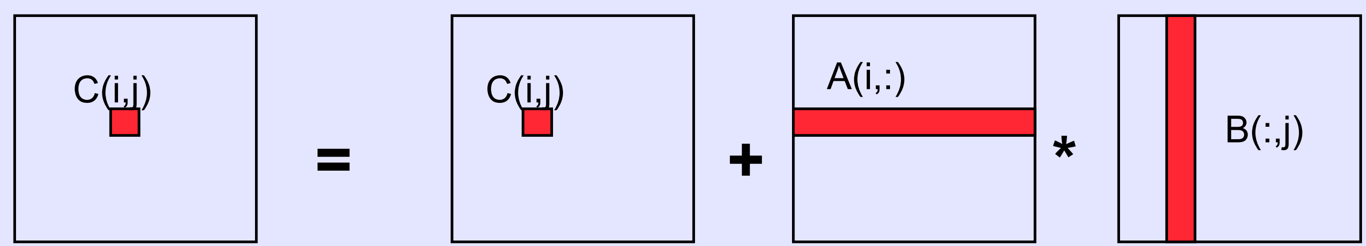
Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **OpenCL overview**
- **Exploring the spec through a series of examples**
  - **Vector addition:**
    - *the basic platform layer*
  - **Matrix multiplication:**
    - *writing simple kernels*
  - **Optimizing matrix multiplication:**
    - *work groups and the memory model*
- **A survey of OpenCL 1.1**

Optimizing matrix multiplication

- 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 assure that for every memory movement, 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.

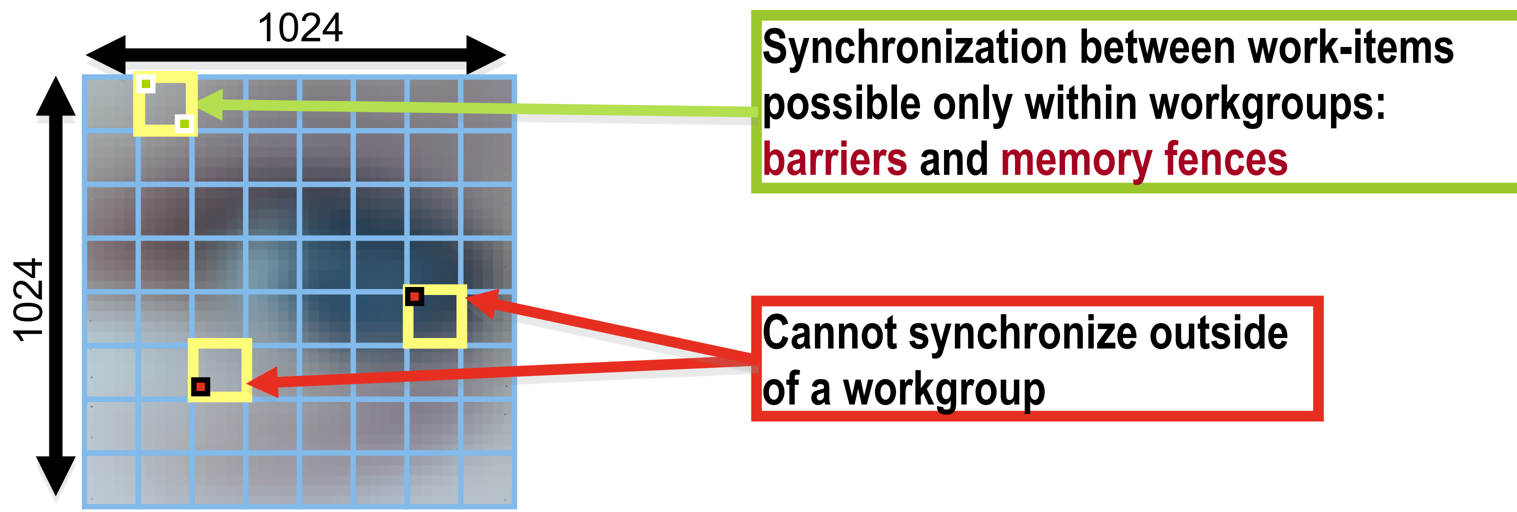


Dot product of a row of A and a column of B for each element of C

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

An N-dimension domain of work-items

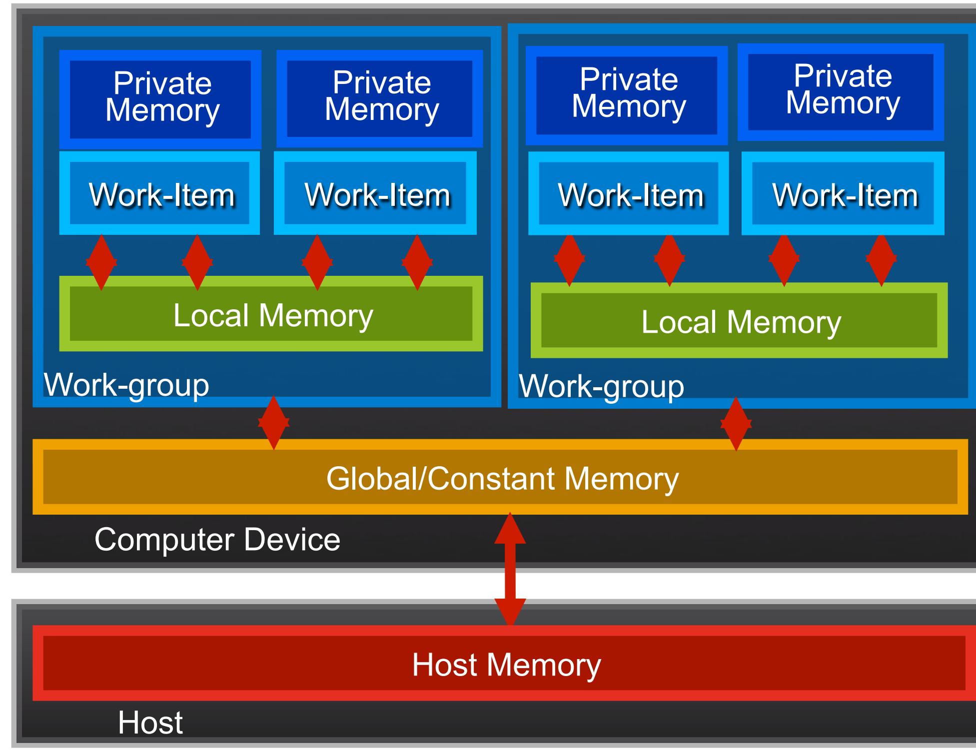
- **Global Dimensions: 1024 x 1024 (whole problem space)**
- **Local Dimensions: 128 x 128 (work group … executes together)**



- **Choose the dimensions that are "best" for your algorithm**

OpenCL memory model

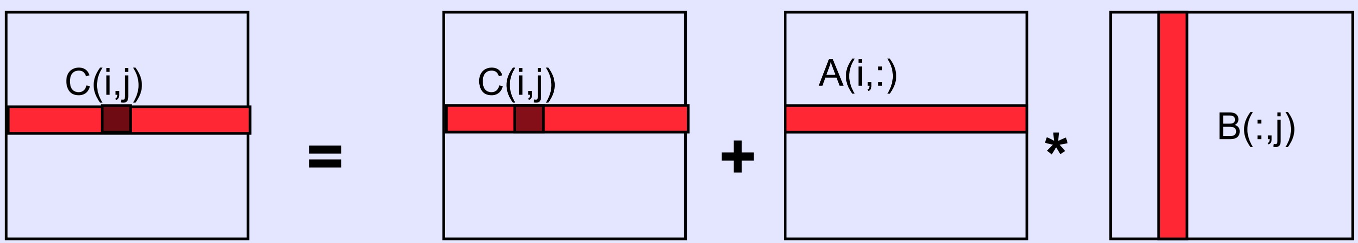
- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a work-group
- **Local Global/Constant Memory**
  - Visible to all work-groups
- **Host Memory**
  - On the CPU



- **Memory management is explicit:**
  You must move data from host -> global -> local *and* back

Optimizing matrix multiplication

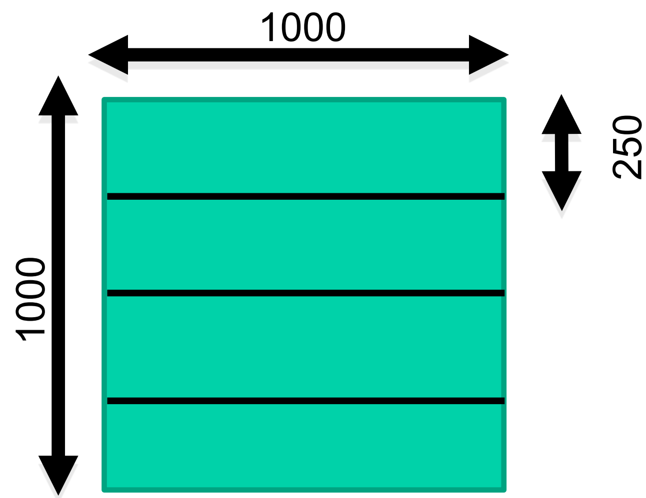
- There may be significant overhead to manage work-items and work-groups.
- So let's have each work-item compute a full row of C



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

An N-dimension domain of work-items

- **Global Dimensions: 1000 x 1000 Whole problem space (*index space*)**
- **Local Dimensions: 250 x 1000 One work group per compute unit**



- **Important implication: there will be a lot fewer work-items (1000 rather than 1000x1000). Why might this matter?**

Reduce work-item overhead …

do one row of C per work-item

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

MatMult host program: one row of C per work-item

Changes to host program:

1. 1D ND Range set to number of rows in the C matrix
2. Local Dim set to 250 so number of work-groups match number of compute units (4 in this case) for our order 1000 matrices

```
global[0] = (size_t) Ndim; local[0] = (size_t) 250;   *ndim = 1;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, local, 0, NULL, NULL);
```

```
#include
int main(
{
   float *A
   int Mdi
   int err,
   size_t g
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
   cl_command_queue commands;
   cl_program program;
   cl_kernel kernel;
   cl_uint nd;
   cl_mem a_in, b_in, c_out;

   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
   err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
   context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

   ...);
   err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);

   *program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource,  NULL, &err);
   err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

   *kernel = clCreateKernel(*program, "mmul", &err);
   err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
   err |= clSetKernelArg(*kernel, 1, sizeof(int), &Ndim);
   err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);

   global[0] = (size_t) Ndim; local[0] = (size_t) 250; *ndim = 1;
   err = clEnqueueNDRangeKernel(commands, kernel, ndim, NULL, global, local, 0, NULL, NULL);
   clFinish(commands);
   err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
   test_results(A, B, c_out);
}
```

Results: MFLOPS

- Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.

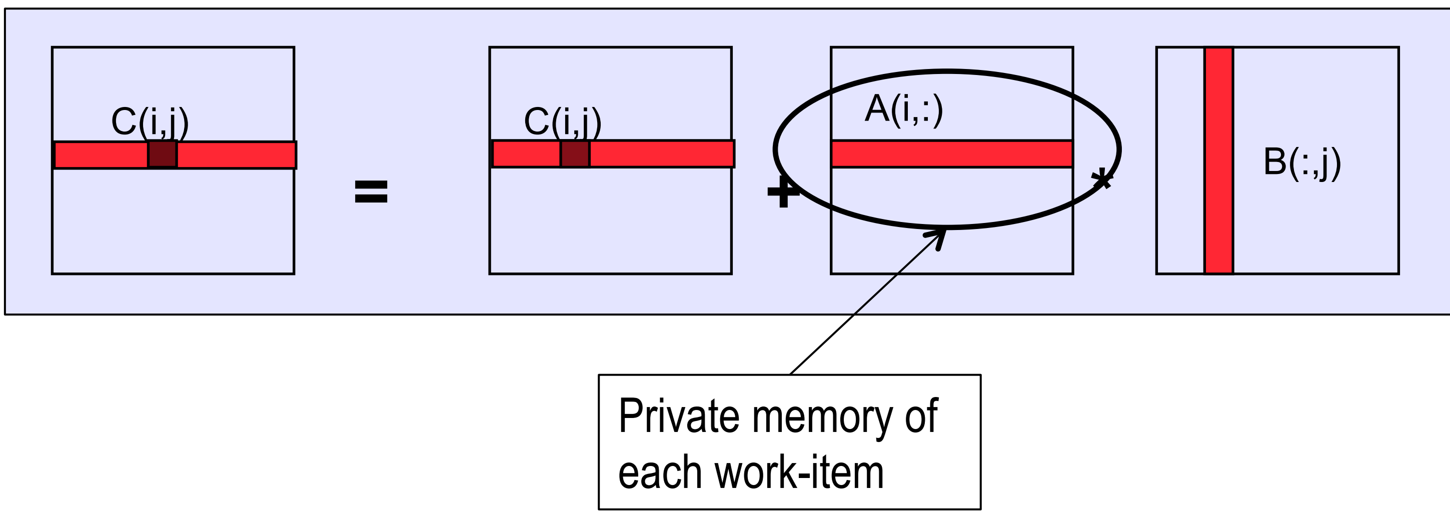
Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
CPU: C(i,j) per work item	744

This on its own didn't help. (← 258)

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Optimizing matrix multiplication

- Notice that each element of C in a row uses 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.



Row of C per work-item, A row private

```
__kernel void mmul(
    const int Mdim,
    const int Ndim,
    const int Pdim,
    __global float* A,
    __global float* B,
    __global float* C)
{
   int k,j;
   int i = get_global_id(0);
   float Awrk[1000];
   float tmp;

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

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

(Actually, this is using ***far*** too much private memory and so Awrk[] will likely be spilled to global memory)

Matrix multiplication performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

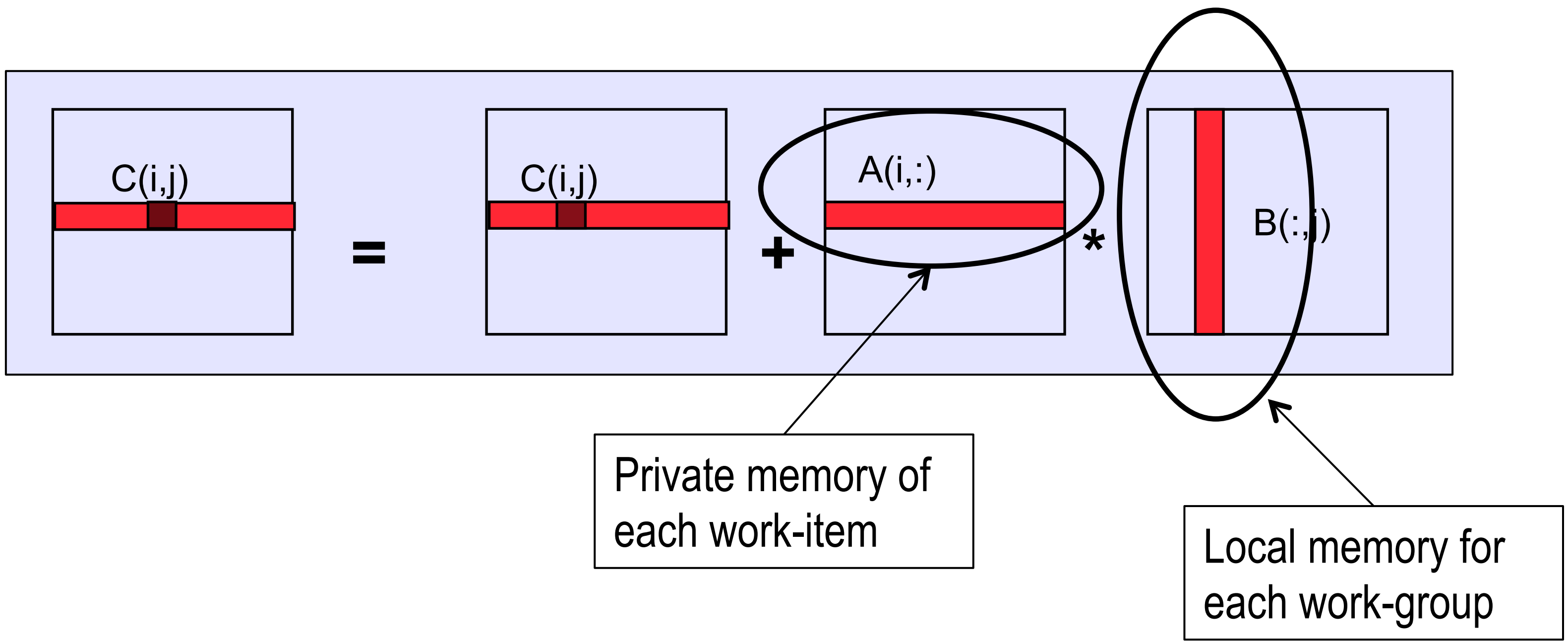
Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
GPU: C row per work item, A row private	873
CPU: C(i,j) per work item	744

Big impact!

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Optimizing matrix multiplication

- We already noticed that each element of C 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 (shared by WIs in a WG)



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

```
__kernel void mmul(
    const int Mdim,
    const int Ndim,
    const int Pdim,
    __global float* A,
    __global float* B,
    __global float* C,
    __local float* Bwrk)
{
   int k,j;
   int i = get_global_id(0);
   int iloc = get_local_id(0);
   int nloc = get_local_size(0);
   float Awrk[1000];
   float tmp;
   for (k=0; k<Pdim; k++)
      Awrk[k] = A[i*Ndim+k];
   for (j=0; j<Mdim; j++){
      for (k=iloc; k<Pdim; k=k+nloc)
         Bwrk[k] = B[k*Pdim+j];
      barrier(CLK_LOCAL_MEM_FENCE);
      tmp = 0.0f;
      for (k=0; k<Pdim; k++)
         tmp += Awrk[k] * Bwrk[k];
      C[i*Ndim+j] += tmp;
   }
```

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

MatMult host program: small change

Changes to host program:

1. Pass local memory to kernels. This requires a change to the kernel argument list … a new call to clSetKernelArg is needed.

This call passes in a pointer to this many bytes of reserved local memory

```
err |= clSetKernelArg(*kernel, 6, sizeof(float)*Pdim, NULL);
```

```
#include
int main(
{
  float *A
  int Mdi
  int err, szA, szB, szC;
  size_t global[DIM];
  size_t local[DIM];
  cl_devic
  cl_conte
  cl_command_queue commands;
  cl_program program;
  cl_kernel kernel;
  cl_uint nd;
  cl_mem a_in, b_in, c_out
  Ndim = ORDER;
  Pdim = ORDER;
  Mdim = ORDER;
  szA = Ndim*Pdim;
  szB = Pdim*Mdim;
  szC = Ndim*Mdim;
  A = (float *)malloc(szA*sizeof(float));
  B = (float *)malloc(szB*sizeof(float));
  C = (float *)malloc(szC*sizeof(float));
  initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
  err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
  context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

  err = clEnqueueWriteBuffer(commands, a_in, CL_TRUE, 0, sizeof(float) * szA, A, 0, NULL, NULL);
  err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);

  e, NULL, &err);
  err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

  *kernel = clCreateKernel(*program, "mmul", &err);

  err |= clSetKernelArg(*kernel, 2, sizeof(int), &Pdim);
  err |= clSetKernelArg(*kernel, 3, sizeof(cl_mem), &a_in);
  err |= clSetKernelArg(*kernel, 4, sizeof(cl_mem), &b_in);
  err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);
  err |= clSetKernelArg(*kernel, 6, sizeof(float)*Pdim, NULL);

  global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *ndim = 1;
  err = clEnqueueNDRangeKernel(commands, kernel, ndim, NULL, global, local, 0, NULL, NULL);
  clFinish(commands);
  err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
  test_results(A, B, c_out);
}
```

Matrix multiplication performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
GPU: C row per work item, A row private	873
GPU: C row per work item, A private, B local	2,472
CPU: C(i,j) per work item	744

Biggest impact so far!

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Matrix multiplications performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	Speedup
CPU: Sequential C (not OpenCL)	1
GPU: C(i,j) per work item, all global	3
GPU: C row per work item, all global	1.5
GPU: C row per work item, A row private	5.2
GPU: C row per work item, A private, B local	15
CPU: C(i,j) per work item	4.5

Wow!!! OpenCL on a GPU is radically faster that C on a CPU, right?

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz