Lecture 5

INTRODUCTION TO OPENCL KERNEL PROGRAMMING

OpenCL C for Compute Kernels

- Derived from ISO C99
  - A few ***restrictions***: no recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported (#include etc.)
- Built-in data types
  - Scalar and vector data types, pointers
  - Data-type conversion functions:
    - convert_type<_sat><_roundingmode>
  - Image types:
    - image2d_t, image3d_t and sampler_t

OpenCL C for Compute Kernels

- Built-in functions — *mandatory*
  - Work-Item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
  - printf (v1.2 only, so not currently for NVIDIA GPUs)
- Built-in functions — *optional* (called “extensions”)
  - Double precision, atomics to global and local memory
  - Selection of rounding mode, writes to image3d_t surface

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

OpenCL C Language Restrictions

- Pointers to functions are ***not*** allowed
- Pointers to pointers allowed ***within*** a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are optional in OpenCL v1.1, but the key word is reserved

  (note: most implementations support double)

Worked example: 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.

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

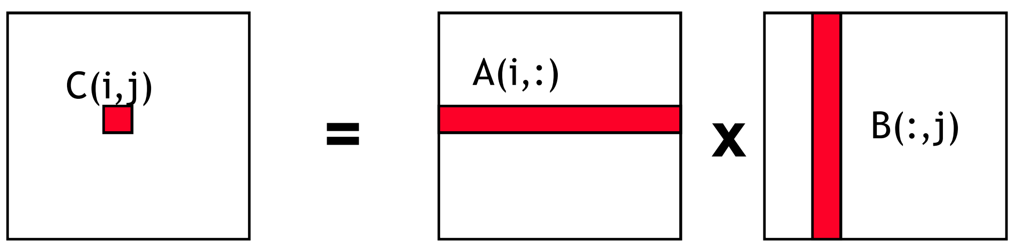
- We solve for x, given a problem Ax=b
  - Ax=b LUx=b
  - $Ux=(L^{-1})b$ $x = (U^{-1})(L^{-1})b$

So we need to be able to do matrix multiplication

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, j) = sum(over k) A(i,k) * B(k,j)
                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.

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

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 co-ordinates

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                          s in each matrix
  double start_                         iming data
  cl::Program p

  Ndim = Pdim =
  szA = Ndim*P
  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

```
  // Compile for first kernel to setup program
  program =                            Source, true);
  Context co                           AULT);
  cl::Comman
  std::vecto
      contex                           CES>();
  cl::Device
  std::string s =
      device.getInfo<CL_DEVICE_NAME>();
  std::cout << "\nUsing OpenCL Device "
           << s << "\n";
```

Setup the platform and build program

```
// Set                                  s,
// and
initma                                  ;
cl::Bu                                  end(), true);
cl::Bu                                  end(), true);
cl::Bu
                    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

```
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.

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

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.

Exercise 6: Matrix Multiplication

- Goal:
  - To write your first complete OpenCL kernel “from scratch”
  - To multiply a pair of matrices
- Procedure:
  - Start with the provided matrix multiplication OpenCL host program including the function to generate matrices and test results
  - Create a kernel to do the multiplication
  - Modify the provided OpenCL host program to use your kernel
  - Verify the results
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result
  - Report the runtime and the MFLOPS

Lecture 6

UNDERSTANDING THE OPENCL MEMORY HIERARCHY