OpenCL
An Introduction for programmers

Benedict Gaster, AMD
Tim Mattson, Intel
Preliminaries:

• Disclosures
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are **not** speaking for our employers.
    - We are **not** speaking for Khronos

• We take our tutorials VERY seriously:
  - Help us improve … give us feedback and tell us how you would make this tutorial even better.
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
It’s a Heterogeneous world

• A modern platform
  Includes:
  – One or more CPUs
  – One or more GPUs
  – DSP processors
  – … other?

OpenCL lets Programmers write a single **portable** program that uses **ALL** resources in the heterogeneous platform

GMCH = graphics memory control hub,  ICH = Input/output control hub
Microprocessor trends

Individual processors are many core (and often heterogeneous) processors.

The Heterogeneous many-core challenge: How are we to build a software ecosystem for the Heterogeneous many core platform?

3rd party names are the property of their owners.
Industry Standards for Programming Heterogeneous Platforms

CPUs
- Multiple cores driving performance increases
- Multi-processor programming – e.g. OpenMP

GPUs
- Increasingly general purpose data-parallel computing
- Graphics APIs and Shading Languages

Emerging Intersection

OpenCL
- Heterogeneous Computing

OpenCL – Open Computing Language
Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors
The **BIG** idea behind OpenCL

- Replace loops with functions (a **kernel**) executing at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

### Traditional loops

```c
void trad_mul(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

```c
kernel void dp_mul(
    global const float *a, 
    global const float *b, 
    global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over “n” work-items
```
The origins of OpenCL

AMD

ATI

Nvidia

Intel

Apple

Merged, needed commonality across products

GPU vendor - wants to steal market share from CPU

CPU vendor - wants to steal market share from GPU

was tired of recoding for many core, GPUs. Pushed vendors to standardize.

Wrote a rough draft straw man API

Khronos Compute group formed

Dec 2008

Third party names are the property of their owners.
OpenCL Working Group within Khronos

• Diverse industry participation …
  - Processor vendors, system OEMs, middleware vendors, application developers.

• OpenCL became an important standard “on release” by virtue of the market coverage of the companies behind it.
OpenCL Timeline

- Launched Jun’08 … 6 months from “strawman” to OpenCL 1.0.
- Rapid innovation to match pace of HW innovation
  - 18 months from 1.0 to 1.1
  - OpenCL 1.2 expected around Dec’12
  - We are committed backwards compatibility .. To protect your SW investments.

During 2H09
Multiple conformant implementations ship across a diverse range of platforms.

- Dec08
  - Khronos publicly releases OpenCL 1.0 specification

- Jun10
  - Khronos publicly releases OpenCL 1.1 specification.
  - Conformant implementations available shortly thereafter

- Jan11
  - Feature freeze on OpenCL 1.2
OpenCL: From cell phone to supercomputer

• OpenCL Embedded profile for mobile and embedded silicon
  - Relaxes some data type and precision requirements
  - Avoids the need for a separate “ES” specification
• Khronos APIs provide computing support for imaging & graphics
  - Enabling advanced applications in, e.g., Augmented Reality
• OpenCL will enable parallel computing in new markets
  - Mobile phones, cars, avionics

A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet.
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
OpenCL Platform Model

- One **Host** + one or more **Compute Devices**
  - Each Compute Device is composed of one or more **Compute Units**
    - Each Compute Unit is further divided into one or more **Processing Elements**
Contexts

- **Context:**
  - The environment within which the *kernels execute and the domain in which* synchronization and memory management is defined. The *context includes a set of devices, the memory accessible to those devices, the corresponding memory properties and one or more command-queues used to schedule execution of a kernel(s) or operations on memory objects.*

- **Contexts are used to contain and manage the state of the “world” in OpenCL.**
  - Kernel execution commands
  - Memory commands - transfer or mapping of memory object data
  - Synchronization commands - constrains the order of commands
Command-Queues

- All commands to a device are submitted through a command-queue.
- Execution of the command-queue is guaranteed to be completed at sync points.
- Each Command-queue points to a single device within a context.
- A single device can simultaneously be pointed to be multiple command queues.
- Multiple command-queues can be created to handle independent commands that don’t require synchronization.

Commands Queued in-order. Can execute in-order or out-of order depending on the queue.
Execution model (kernels)

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

```c
kernel void square(
    global float* input,
    global float* output)
{
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

```
<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 1 0 9 2 4 1 1 9 7 6 1 2 2 1 9 8 4 1 9 2 0 0 7 8</td>
<td>36 1 1 0 81 4 16 1 1 81 49 36 1 4 4 1 81 64 16 1 81 4 0 0 49 64</td>
</tr>
</tbody>
</table>
```
An N-dimension domain of work-items

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

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

Cannot synchronize outside of a workgroup

Choose the dimensions that are “best” for your algorithm
Keeping track of work-items and work-groups

- get_work_dim = 1
- get_global_size = 26
- get_num_groups = 2
- get_group_id = 0
- get_local_size = 13
- get_local_id = 8
- get_global_id = 21

Workgroups:

Input:

```
[6 1 1 0 9 2 4 1 1 9 7 6 1 2 2 1 9 8 4 1 9 2 0 0 7 8]
```
Kernel Execution

• A command to execute a kernel must be enqueued to the command-queue
  - `clEnqueueNDRangeKernel()`
    - Data-parallel execution model
    - Describes the *index space* for kernel execution
    - Requires information on NDRange dimensions and work-group size
  - `clEnqueueTask()`
    - Task-parallel execution model (multiple queued tasks)
    - Kernel is executed on a single work-item
  - `clEnqueueNativeKernel()`
    - Task-parallel execution model
    - Executes a native C/C++ function not compiled using the OpenCL compiler
    - This mode does not use a kernel object so arguments must be passed in
OpenCL Memory Model

• **Private Memory**
  - Per work-item

• **Local Memory**
  - Shared within a workgroup

• **Local Global/Constant Memory**
  - Visible to all workgroups

• **Host Memory**
  - On the CPU

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

• **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**

• **Consistency of memory shared between commands (e.g. kernel invocations) are enforced through synchronization (events)**
Building Program objects

• The program object encapsulates:
  - A context
  - The program source/binary
  - List of target devices and build options

• The Build process … to create a program object
  - `clCreateProgramWithSource()`
  - `clCreateProgramWithBinary()`

```
kernel void horizontal_reflect(read_only image2d_t src, write_only image2d_t dst)
{
    int x = get_global_id(0);  // x-coord
    int y = get_global_id(1);  // y-coord
    int width = get_image_width(src);
    float4 src_val = read_imagef(src, sampler,
        (int2)(width-1-x, y));
    write_imagef(dst, (int2)(x, y), src_val);
}
```
OpenCL C for Compute Kernels

• Derived from ISO C99
  - A few restrictions: recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported

• 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

• Built-in Functions — Required
  - work-item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions

• Built-in Functions — Optional
  - 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
  - Kernels can call other kernel functions

• Address space qualifiers
  - __global, __local, __constant, __private
  - Pointer kernel arguments must be declared with an address space qualifier

• Work-item functions
  - Query work-item identifiers
    - 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
• Bit-fields are not supported
• Variable length arrays and structures are not supported
• Recursion is not supported
• Writes to a pointer of types less than 32-bit are not supported
• Double types are not supported, but reserved
Vector Types

• Portable between different vector instruction sets
• Vector length of 2, 4, 8, and 16
• char2, ushort4, int8, float16, double2, ...
• Endian safe
• Aligned at vector length
• Vector operations (elementwise) and built-in functions
Vector Operations

• Vector literal

```cpp
int4 vi0 = (int4) -7;

int4 vi1 = (int4)(0, 1, 2, 3);
```

• Vector components

```cpp
vi0.lo = vi1.hi;

int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
```

• Vector ops

```cpp
vi0 += vi1;

vi0 = abs(vi0);
```
OpenCL summary

Context

CPU

GPU

Programs

Kernels

Memory Objects

Command Queues

Compile code

Create data & arguments

Send to execution

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

dp_mul
CPU program binary

dp_mul
GPU program binary

Images

Buffers

In Order Queue

Out of Order Queue

Compute Device

Programs

Kernels

Memory Objects

Command Queues
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
Example: vector addition

• The “hello world” program of data parallel programming is a program to add two vectors

\[ C[i] = A[i] + B[i] \quad \text{for } i=1 \text{ to } N \]

• For the OpenCL solution, there are two parts
  - Kernel code
  - Host code
__kernel void vec_add (__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

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

• 5 simple steps in a basic Host program
  1. Define the platform … platform = devices+context+queues
  2. Create and Build the program (dynamic library for kernels)
  3. Setup memory objects
  4. Define kernel (attach kernel function to arguments)
  5. Submit commands … move memory objects and execute kernels

Our goal is extreme portability so we expose everything (i.e. we are a bit verbose). But most of a host code is the same from one application to the next … the re-use makes the verbosity a non-issue
1. Define the platform

• Grab the first available Platform:
  \[
  \text{err = clGetPlatformIDs}(1, \text{&firstPlatformId}, \text{&numPlatforms});
  \]

• Use the first GPU device the platform provides
  \[
  \text{err = clGetDeviceIDs(firstPlatformId, CL\_DEVICE\_TYPE\_GPU, 1, \text{&device_id}, \text{NULL});}
  \]

• Create a simple context with a single device
  \[
  \text{context = clCreateContext}(0, 1, \text{&device_id}, \text{NULL}, \text{NULL}, \text{&err});
  \]

• Create a simple command queue to feed our compute device
  \[
  \text{commands = clCreateCommandQueue(context, device_id, 0, \text{&err});}
  \]
2. Create and Build the program

- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (common in real apps).

- Build the program object:

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

- Compile the program to create a “dynamic library” from which specific kernels can be pulled:

```c
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

- Fetch and print error messages (if(err != CL_SUCCESS))

```c
size_t len;           char buffer[2048];
clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer),
                        buffer, &len);
printf("%s\n", buffer);
```
3. Setup Memory Objects

• For vector addition, 3 memory objects ... one for each input vector (A and B) and one for the output vector (C).

• Create input vectors and assign values on the host:

```c
float a_data[LENGTH], b_data[LENGTH], c_res[LENGTH];
for(i = 0; i < count; i++){
    a_data[i] = rand() / (float)RAND_MAX;
    b_data[i] = rand() / (float)RAND_MAX;
}
```

• Connect host data to OpenCL memory objects

```c
a_in = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, NULL);
b_in = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, NULL);
c_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, NULL);
```
4. Define the kernel

• Create kernel object from the kernel function “vadd”

```
kernel = clCreateKernel(program, "vadd", &err);
```

• Attach arguments to the kernel function “vadd” to memory objects

```
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_in);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_in);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_out);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
```
5. Submit commands

- Write Buffers from host into global memory (as blocking operations)
  
  ```
  err = clEnqueueWriteBuffer(commands, a_in, CL_TRUE, 0,
                            sizeof(float) * count, a_data, 0, NULL, NULL);
  err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0,
                            sizeof(float) * count, b_data, 0, NULL, NULL);
  ```

- Enqueue the kernel for execution
  
  ```
  err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL,
                                &global, &local, 0, NULL, NULL);
  ```

- What until all previous commands are done, then read back the result.
  
  ```
  clFinish(commands);
  err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0,
                            sizeof(float) * count, c_res, 0, NULL, NULL);
  ```
Vector Addition - Host Program

// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
~CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
sizeof(cl_float)*n, NULL,
NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
&program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
NULL);

// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
sizeof(cl_mem));

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
Vector Addition - Host Program

// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
    CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

Define platform and queues
// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
    NULL, &cb); devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
    devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
    0, NULL);

Define Memory objects
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
    NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
    NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(cl_float)*n, NULL,
    NULL);

Create the program
// create the program
program = clCreateProgramWithSource(context, 1,
    &program_source, NULL, NULL);

Build the program
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
    NULL);

Create and setup kernel
// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
    sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
    sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
    sizeof(cl_mem));

Execute the kernel
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
    NULL, global_work_size, NULL, 0, NULL, NULL);

Read results on the host
// read
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE,
    0, n*sizeof(cl_float), dst, 0, NULL, NULL);

It’s complicated, but most of this is “boilerplate” and not as bad as it looks.
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
  \[\begin{align*}
  x + 2y + z &= 1 \\
  x + 3y + 3z &= 2 \\
  x + y + 4z &= 6
  \end{align*}\]
  - 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}
$$

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

- A matrix, $A \in \mathbb{R}^{M \times P}$ multiplies a vector, $x \in \mathbb{R}^P$ to define a linear transformation of that vector into $\mathbb{R}^m$.
- Matrix multiplication defines the composition of two linear transformations on that vector space

Compute $C = A \times B$ where

- $C \in \mathbb{R}^{M \times N}$
- $A \in \mathbb{R}^{M \times P}$
- $B \in \mathbb{R}^{P \times N}$

- Matrix multiplication is the core building block of Linear Algebra
Matrix Multiplication: Sequential code

```c
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 Multiplications Performance

- Results on an Apple laptop with an Intel CPU.

<table>
<thead>
<tr>
<th>Case</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Sequential C (not OpenCL)</td>
<td>167</td>
</tr>
</tbody>
</table>

Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz
Matrix Multiplication: OpenCL kernel (1/4)

```c
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)

```c
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];
        }
    }
}
}
```

```c
__kernel 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];
            }
        }
    }
}
```

Mark as a kernel function and specify memory qualifiers
Matrix Multiplication: OpenCL kernel (3/4)

```
__kernel 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];
            }
        }
    }
}
```

Remove outer loops and set work item coordinates
Matrix Multiplication: OpenCL kernel (4/4)

```c
__kernel 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 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;
    tmp = 0.0;
    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[2];
    size_t local[2];
    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, nd, 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);
}

- Page 51
Matrix Multiplication host program

```c
#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[2];
    size_t local[2];
    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);
    error = 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);
    err = clEnqueueNDRangeKernel(commands, kernel, 2, 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 might 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

```c
#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 ndim;
    cl_mem a_in, b_in, c_out;

    Mdim = ORDER;
    Ndim = ORDER;
    Pdim = 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);

    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);

    *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, 2, 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);
}
```

**Setup the platform**

```c
erroff = 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 buffers and write A and B matrices to the device memory**

```c
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);
```

**Build the program, define the Kernel, and setup arguments**

```c
*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);
```

**Run the kernel and collect results**

```c
err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, global, NULL, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL);
```
Matrix Multiplication host program

```c
#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;

    Mdim = ORDER;
    Pdim = ORDER;
    Ndim = 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, nd, NULL, global, NULL, 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);
}
```

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 for us.
Matrix Multiplications Performance

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

<table>
<thead>
<tr>
<th>Case</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Sequential C (not OpenCL)</td>
<td>167</td>
</tr>
<tr>
<td>GPU: C(i,j) per work item, all global</td>
<td>511</td>
</tr>
<tr>
<td>CPU: C(i,j) per work item, all global</td>
<td>744</td>
</tr>
</tbody>
</table>

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

3rd party names are the property of their owners.
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:
  - $2n^3 = O(n^3)$ Flops
  - operates on $3n^2$ 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.

\[
C(i,j) = C(i,j) + A(i,:) \ast B(:,j)
\]

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)

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

Cannot synchronize outside of a workgroup

• Choose the dimensions that are “best” for your algorithm
OpenCL Memory Model

- **Private Memory**
  - Per work-item

- **Local Memory**
  - Shared within a workgroup

- **Local Global/Constant Memory**
  - Visible to all workgroups

- **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.
- Let’s have each work item compute a full row of $C$

$$C(i, j) = C(i, j) + A(i,:) * B(:, j)$$

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)
- Local Dimensions: 250 x 1000 (One work group per compute unit)
Reduce work-item overhead …

do one row of C per work-item

```c
__kernel 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++){
        tmp = 0.0;
        for(k=0;k<Pdim;k++)
            tmp += A[i*Ndim+k] * B[k*Pdim+j];
        C[i*Ndim+j] = tmp;
    }
}
```
Changes to host program:
1. 1 D 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

```c
#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[4];
  size_t local[4];
  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;

  Mdim = ORDER;
  Pdim = ORDER;
  Ndim = 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 = clEnqueueNDRangeKernel(commands, kernel, 4, NULL, global, local, 0, NULL, NULL);

  err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);
  err = clEnqueueWriteBuffer(commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL);
  test_results(A, B, c_out);

  return 0;
}
```
Results: MFLOPS

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

<table>
<thead>
<tr>
<th>Case</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Sequential C (not OpenCL)</td>
<td>167</td>
</tr>
<tr>
<td>GPU: C(i,j) per work item, all global</td>
<td>511</td>
</tr>
<tr>
<td>GPU: C row per work item, all global</td>
<td>258</td>
</tr>
<tr>
<td>CPU: C(i,j) per work item</td>
<td>744</td>
</tr>
</tbody>
</table>

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

This on its own didn’t help.
Optimizing Matrix Multiplication

- Notice that each element of C in a row uses the same row of A.
- Let’s copy A into private memory so we don’t incur the overhead of pulling it from global memory for each C(i,j) computation.
__kernel 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 = 0.0;
        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 from global memory before we start with the matrix multiplications.
MatMult host program: one row of C per work-item

Changes to host program:
1. 1 D 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

```c
#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);

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

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

<table>
<thead>
<tr>
<th>Case</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Sequential C (not OpenCL)</td>
<td>167</td>
</tr>
<tr>
<td>GPU: C(i,j) per work item, all global</td>
<td>511</td>
</tr>
<tr>
<td>GPU: C row per work item, all global</td>
<td>258</td>
</tr>
<tr>
<td>GPU: C row per work item, A row private</td>
<td>873</td>
</tr>
<tr>
<td>CPU: C(i,j) per work item</td>
<td>744</td>
</tr>
</tbody>
</table>

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

- Notice that each element of C uses the same row of A.
- Each work-item in a work-group uses the same columns of B.
- Let’s store the B columns in local memory.
Row of C per work item, A row private, B columns local

__kernel 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 = 0.0;
        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.
#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 = malloc(szA* sizeof(float));
    B = malloc(szB* sizeof(float));
    C = 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);
    err |= clSetKernelArg(*kernel, 6, sizeof(float)*Pdim, NULL);

    err = clEnqueueNDRangeKernel(commands, kernel, 1, 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);
}

Changes to host program:
1. Modify so we pass local memory to kernels. This requires a change to the kernel argument list ... a new call to clSetKernelArg is needed.

Changes to host program:

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

*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, 6, sizeof(float)*Pdim, NULL);

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 = 2;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, local, 0, NULL, NULL);
cFinish(commands);
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL );
}test_results(A, B, c_out);
Matrix Multiplications Performance

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

<table>
<thead>
<tr>
<th>Case</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Sequential C (not OpenCL)</td>
<td>167</td>
</tr>
<tr>
<td>GPU: C(i,j) per work item, all global</td>
<td>511</td>
</tr>
<tr>
<td>GPU: C row per work item, all global</td>
<td>258</td>
</tr>
<tr>
<td>GPU: C row per work item, A row private</td>
<td>873</td>
</tr>
<tr>
<td>GPU: C row per work item, A private, B local</td>
<td>2472</td>
</tr>
<tr>
<td>CPU: C(i,j) per work item</td>
<td>744</td>
</tr>
</tbody>
</table>

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

3rd party names are the property of their owners.
Matrix Multiplications Performance

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

<table>
<thead>
<tr>
<th>Case</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Sequential C (not OpenCL)</td>
<td>1</td>
</tr>
<tr>
<td>GPU: C(i,j) per work item, all global</td>
<td>3</td>
</tr>
<tr>
<td>GPU: C row per work item, all global</td>
<td>1.5</td>
</tr>
<tr>
<td>GPU: C row per work item, A row private</td>
<td>5.2</td>
</tr>
<tr>
<td>GPU: C row per work item, A private, B local</td>
<td>15</td>
</tr>
<tr>
<td>CPU: C(i,j) per work item</td>
<td>4.5</td>
</tr>
</tbody>
</table>

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

3rd party names are the property of their owners.
**CPU vs GPU: Let’s be fair**

- We made no attempt to optimize the CPU/C code but we worked hard to optimize OpenCL/GPU code.

- **Lets optimize the CPU code**
  - Use compiler optimization (level O3).
  - Replace float with double (CPU ALU’s like double)
  - Reorder loops:

```c
void mat_mul_ijk(int Mdim, int Ndim, int Pdim,
                 double *A, double *B, double *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*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
}
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Float, no opt</td>
<td>167 mflops</td>
</tr>
<tr>
<td>-</td>
<td>Double, O3</td>
<td>272 mflops</td>
</tr>
</tbody>
</table>

- ij: 272 mflops
- ikj: 1130 mflops
- kij: 481 mflops
Matrix Multiplications Performance

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

<table>
<thead>
<tr>
<th>Case</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Sequential C (not OpenCL)</td>
<td>1</td>
</tr>
<tr>
<td>GPU: C(i,j) per work item, all global</td>
<td>0.45</td>
</tr>
<tr>
<td>GPU: C row per work item, all global</td>
<td>0.23</td>
</tr>
<tr>
<td>GPU: C row per work item, A row private</td>
<td>0.77</td>
</tr>
<tr>
<td>GPU: C row per work item, A private, B local</td>
<td>2.2</td>
</tr>
<tr>
<td>CPU: C(i,j) per work item</td>
<td>0.66</td>
</tr>
</tbody>
</table>

And we still are only using one core … and we are not using SSE so there is lots of room to further optimize the CPU code.

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
OpenCL 1.1 - API

• Thread-safety
  • All API calls, except `clSetKernelArg`, are thread safe
• Sub-buffer objects
  • Create an object that represents a specific region in a buffer object
  • Easy and efficient mechanism to distribute regions of a buffer object across multiple devices
  • OpenCL™ synchronization mechanism ensures modifications to sub-buffer object reflected in appropriate region of parent buffer object
OpenCL 1.1 - API

• User Events
  • `clEnqueue***` commands can wait on event
  • In OpenCL™ 1.0, events can only refer to OpenCL™ commands
  • Need ability to enqueue commands that wait on an external, user defined, event

• Event CallBacks
  • `clSetEventCallbackFn` to register a user callback function
    • called when command identified by event has completed
    • Allows applications to enqueue new OpenCL™ commands based on event state changes in a non-blocking manner
  • Lot’s more API stuff too
OpenCL 1.1 - Language

- Implicit Conversions
  - OpenCL™ 1.0 requires widening for arithmetic operators
    ```
    float4 a, b;
    float c;
    b = a + c;  // c is widened to a float4 vector
      // first and then the add is performed
    ```
  - OpenCL™ 1.1 extends this feature for all operators
    - relational, equality, bitwise, logical, ternary
OpenCL 1.1 - Language

• 3-component vector data types
  • And everyone applauds....well almost everyone

• cl_khr_byte_addressable as core feature

• Atomic extensions are now core features
  • cl_khr_global_int32_{base | extended}_atomics
  • cl_khr_local_int32_{base | extended}_atomics
OpenCL 1.1 - Language

• New built-in functions
  • get_global_offset
  • clamp for integer data types
  • async_work_group_strided_copy
  • strided async copy of data from global <--- local memory
• shuffle - construct a permutation of elements from 1 or 2 input vectors and a mask
OpenCL 1.1 – OpenCL/OpenGL Sharing

- Improve performance of OpenCL/ OpenGL interoperability
  - Portable OpenCL/ OpenGL sharing requires
    - a `glFinish` before `clEnqueueAcquireGLOObjects`
    - a `clFinish` after `clEnqueueReleaseGLOObjects`
  - `glFinish / clFinish` are heavyweight APIs
OpenCL 1.1 – OpenCL/OpenGL Sharing

• Improve performance of OpenCL/OpenGL interoperability
  • Create a OpenCL event from an OpenGL sync object
  • Create a OpenGL sync object from a OpenCL event
  • Allows for a finer grained waiting mechanism
    • Use `event_wait_list` argument for events that refer to OpenGL commands to complete
    • Use OpenGL sync APIs to wait for specific OpenCL™ commands to complete
Conclusion

- 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 only if its used correctly:
  - Implicit SIMD data parallel code has the best chance of mapping onto a diverse range of hardware ... once OpenCL implementation quality catches up with mature shader languages.

- The future is clear:
  - Braided parallelism mixing task parallel and data parallel code in a single program ... balancing the load among ALL OF the platform’s available resources.