Advanced Topics in Heterogeneous Programming with OpenCL

Ben Gaster  AMD  benedict.gaster @amd.com
Tim Mattson  Intel  timothy.g.mattson @intel.com
Ian Buck  NVIDIA  ibuck @nvidia.com
Mike Houston  AMD  Michael.Houston @amd.com
Peng Wang  NVIDIA  penwang @nvidia.com
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

• Review of basic constructs in OpenCL, 20 min
  – A quick revision of core OpenCL constructs
  – The future of OpenCL

• Examples (Part 1):
  – Conjugate gradient solver: sparse-matrix vector multiply kernel, 30 min
  – Device Fission and vector containers, 30 min

• Examples (Part 2):
  – N-body kernels, 30 min
  – FFT and wavelets, 30 min
  – Bilateral Gaussian, 15 – 30 min (depending on time)
Quick review of core OpenCL constructs

Benedict R. Gaster
AMD
It’s a Heterogeneous World

• A modern platform
  Includes:
  – One or more CPUs
  – One or more GPUs
  – Accelerators such as DSPs

GMCH = graphics memory control hub
ICH = Input/output control hub
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

![Diagram of OpenCL Platform Model](image)
OpenCL Execution Model

- An OpenCL application runs on a host which submits work to the compute devices
  - **Work item**: the basic unit of work on an OpenCL device
  - **Kernel**: the code for a work item. Basically a C function
  - **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
  - **Context**: The environment within which work-items executes ... includes devices and their memories and command queues

- Applications queue kernel execution instances
  - Queued in-order ... one queue to a device
  - Executed in-order or out-of-order
The BIG Idea behind OpenCL

- OpenCL execution model ...
  - Define N-dimensional computation domain
  - Execute a kernel at each point in computation domain

**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
```
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

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

Cannot synchronize outside of a workgroup
OpenCL Memory Model

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

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

• **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
Build the Program object

- 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 Code

```c
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);
}
```
Synchronization: Queues & Events

• Each individual queue can execute in-order or out-of-order
  – For in-order queue, all commands execute in order
• You must explicitly synchronize between queues
  – Multiple devices each have their own queue
  – Multiple queues per device
  – Use events to synchronize
• Events
  – Commands return events and obey waitlists
  – `clEnqueue*(..., num_events_in_waitlist, *event_waitlist, *event);`
Programming kernels: OpenCL C

• Derived from ISO C99
  – But without some C99 features such as standard C99 headers, function pointers, recursion, variable length arrays, and bit fields

• Language Features Added
  – Work-items and workgroups
  – Vector types
  – Synchronization
  – Address space qualifiers

• Also includes a large set of built-in functions
  – Image manipulation
  – Work-item manipulation,
  – Math functions, etc.
Programming Kernels: What is a kernel

- A data-parallel function executed by each work-item

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

- get_global_id(0) = 7

<table>
<thead>
<tr>
<th>Input</th>
<th>6 1 1 0 9 2 4 1 1 9 7 6 8 2 2 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>36 1 1 0 81 4 16 1 1 81 49 36 64 4 4 25</td>
</tr>
</tbody>
</table>
Programming Kernels: WorkItems & Groups

get_work_dim = 1

global size = 16

get_num_groups = 2

get_group_id = 0

get_local_size = 8

get_local_id = 3

get_global_id = 11
Programming Kernels: Data Types

• Scalar data types
  – char, uchar, short, ushort, int, uint, long, ulong, float
  – bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)

• Image types
  – image2d_t, image3d_t, sampler_t

• Vector data types
  – Vector lengths 2, 4, 8, & 16 (char2, ushort4, int8, float16, double2, ...)
  – Endian safe
  – Aligned at vector length
  – Vector operations

Double is an optional type in OpenCL 1.0
Programming Kernels: Vector Operations

• Vector Literal

```c
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);
```

• Vector Components

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

• Vector Operations

```c
vi0 += vi1;
```
Introducing OpenCL 1.1

New version of the OpenCL specification adds significant new functionality:

• API and Language Features
• Developer feedback, Ease of use and Performance
• Improved OpenGL Interoperability
OpenCL 1.1 – Thread-safety & Buffers

• Thread-safety
  – All API calls except `clSetKernelArg` are now thread-safe

• Sub-Buffer Objects
  – Easy and efficient way to distribute regions of a buffer across multiple devices
  – Modifications to sub-buffer objects reflected in appropriate regions of parent buffer object.

• Reading, writing & copying rectangular regions in a buffer object
  – Specify the following
    • Region type – 2D or 3D
    • Row-pitch for a 2D & 3D region and Slice-pitch for a 3D region
  – `clEnqueue{Read|Write|Copy}BufferRect`
OpenCL 1.1 – Events

• User Events
  – All clEnqueue* commands take a list of events to wait on
  – In OpenCL 1.0, events can only refer to OpenCL commands
  – User events allow developers to enqueue commands that wait on an external event

• Event Callbacks
  – Allow applications to enqueue CL commands based on event state changes in a non-blocking manner
  – clSetEventCallback to register a user callback function
    • Called when command identified by event has completed
    • Recommend not calling expensive system APIs, OpenCL APIs that create objects or enqueue blocking commands in the callback function.
OpenCL 1.1 – Memory Object Callbacks

• Memory Object Destructor Callback
  – For `cl_mem` objects created with `CL_MEM_USE_HOST_PTR` need a way to determine when it is safe to free or reuse the `host_ptr`
  – Lazy deallocation of `cl_mem` objects make this a little difficult
  – `clSetMemObjectDestructorCallback`
    • Registers a destructor callback function
    • Called when the memory object is ready to be deleted
  – Recommend **not calling** expensive system APIs, OpenCL APIs that create objects or enqueue blocking commands in the callback function.
OpenCL 1.1 – Queries

• Kernel Queries
  – CL_KERNEL_PREFERRED_WORKGROUP_SIZE_MULTIPLE
    • A performance hint

• Device Queries
  – CL_DEVICE_LOCAL_MEM_SIZE
    • Increased from 16 KB to 32 KB
  – CL_DEVICE_MAX_PARAMETER_SIZE
    • Increased from 256 to 1024 bytes
  – CL_DEVICE_OPENCL_C_VERSION
    • Version of OpenCL C supported by device.
  – CL_DEVICE_HOST_UNIFIED_MEMORY
    • Whether device & host have a unified memory subsystem
OpenCL 1.1 – Additional API Features

• global_work_offset
  – Argument to `clEnqueueNDRangeKernel`
  – No longer required to be a NULL value
  – Enqueue kernels that operate on different regions of the N-D range

• C++ API bindings
  – A wrapper API
  – Built on top of the OpenCL 1.1 API specification (not a replacement)
OpenCL 1.1 – Language Features

• Implicit Conversions
  – OpenCL 1.0 requires widening for arithmetic operators

```c
float4 a, b;
float c;

b = a + c; // c is widened to a float4 first
    // and then the + is performed.
```

– OpenCL 1.1 extends this feature to all operators
  • relational, equality, bitwise, logical and ternary
OpenCL 1.1 – Language Features

• 3-component vector data types
  – Useful data type for a number of applications such as game physics
  – Aligned to the corresponding 4-component data type
  – vload3 and vstore3 can be used to view a buffer of scalar elements as a packed buffer of 3-component vectors.

• cl_khr_byte_addressable is a core feature
  – Writes to a pointer or array of type char, char2, uchar, uchar2, short, ushorts and half are now supported.

• 32-bit atomic operations to local and global memory is a core feature
OpenCL 1.1 – Built-in Functions

• **get_global_offset**
  - Global offset values specified to `clEnqueueNDRangeKernel`

• **clamp** for integer data types
  - Only floating-point types were supported in OpenCL 1.0

• **async_work_group_strided_copy**
  - Gather from global to local memory
  - Scatter from local to global memory

• **shuffle**
  - Construct a runtime permutation of elements from 1 or 2 vectors and a mask

```
uint4 mask = (uint4)(3, 2, 1, 0);
float4 a;
float4 r = shuffle(a, mask)
// r.s0123 = a.wzyx
```
OpenCL 1.1 – Images

• Addressing mode – **CL_ADDRESS_MIRRORED_REPEAT**
  – Flip the image coordinate at every integer junction
  – Can only be used with normalized coordinates i.e. **CL_NORMALIZED_COORDS_TRUE** must be set in the sampler

• Optional image formats – **CL_Rx**, **CL_RGx** and **CL_RGBx**
  – Similar to CL_R, CL_RG and CL_RGB except *alpha = 0 at edges*
  – For image processing, *alpha* must always be 0 at edges.
OpenCL 1.1 – OpenCL / OpenGL Sharing

- Improve performance of CL / GL interoperability
  - In OpenCL 1.0, portable CL / GL sharing requires
    - a `glFinish` before `clEnqueueAcquireGLObjects`
    - a `clFinish` after `clEnqueueReleaseGLObjects`
  - `glFinish` and `clFinish` are heavyweight APIs

- New Extensions
  - `cl_khr_gl_event`
    - Create a CL event from a GL sync object
  - `GL_ARB_cl_event`
    - Create a GL sync object from a CL event

- Use CL events and GL sync objects for a faster & finer-grained sharing
**OpenCL 1.1 – OpenCL / OpenGL Sharing**

```c
gl_sync = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
// queue GL commands

gl_event = clCreateEventFromGLSyncKHR(context, gl_sync, NULL);
clEnqueueAcquireGLObjects(queue, num_objects, mem_objects, 1, &gl_event, NULL);
// queue CL commands

clEnqueueReleaseGLObjects(queue, num_objects, mem_objects, 0, NULL, &release_event);

cl_sync = glCreateSyncFromCLEventARB(context, release_event, 0);
glWaitSync(cl_sync, 0, TIMEOUT_IGNORED);
// queue GL commands
```
The Future of OpenCL

• Both OpenCL (and Cuda) are evolving!

• Fast developing language because:
  – Still very young (<5 years even if you include Cuda!)
  – Still understanding heterogeneous, with many-core, programming... BUT

• The age of multi-core and many-core is NOW!
Three Eras of Processor Performance

Single-Core Era

Enabled by:
✓ Moore’s Law
✓ Voltage Scaling
✓ MicroArchitecture

Constrained by:
✗ Power
✗ Complexity

Multi-Core Era

Enabled by:
✓ Moore’s Law
✓ Desire for Throughput
✓ 20 years of SMP arch

Constrained by:
✗ Power
✗ Parallel SW availability
✗ Scalability

Heterogeneous Systems Era

Enabled by:
✓ Moore’s Law
✓ Abundant data parallelism
✓ Power efficient GPUs

Constrained by:
✗ Programming models
✗ Communication overheads
Agenda

• Review of basic constructs in OpenCL, 20 min
  – A quick revision of core OpenCL constructs
  – The future of OpenCL

• Examples (Part 1):
  – Conjugate gradient solver: sparse-matrix vector
    multiply kernel, 30 min
  – Device Fission and vector containers, 30 min

• Examples (Part 2):
  – N-body kernels, 30 min
  – FFT and wavelets, 30 min
  – Bilateral Gaussian, 15 – 30 min (depending on time)
Conjugate Gradient Solver: Sparse Matrix-Vector Multiply Kernel

Peng Wang, Ian Buck
NVIDIA
Overview

• Conjugate gradient solver
• Sparse matrix data structures
• Sparse matrix vector multiply (SpMV) on GPU
• Considerations for linear solvers
Conjugate Gradient Solver

To implement CG on GPU, one need:

- Scalar-vector multiplication
- Vector addition
- Dot product
- SpMV

The only non-trivial part is SpMV
SpMV Overview

• Bases for iterative solvers: Krylov subspace method, multigrid
• Memory-bound: low math-to-load ratio
• Irregular memory access, unlike BLAS
• The best sparse matrix data structure on GPU can be different from CPU
Sparse Matrix Format

(DIA) Diagonal
(ELL) ELLPACK
(CSR) Compressed Row
(HVB) Hybrid
(COO) Coordinate

Structures

Structured

Unstructured
ELLPACK (ELL)

• Storage for K nonzeros per row
  – Pad rows with fewer than K nonzeros
  – Inefficient when row length varies
Hybrid Format

- ELL handles *typical* entries
- COO handles *exceptional* entries
  - Implemented with segmented reduction
__kernel void spmv_csr_scalar(int nRows,
   __global int *rows,
   __global int *cols,
   __global double *vals,
   __global double *x,
   __global double *y)
{
    int row = get_global_id(0);
    if (row < nRows) {
      double sum = 0.0;
      int start = rows[row];
      int end = rows[row+1];
      for (int j = start; j < end; j++)
        sum += vals[j]*x[cols[j]];
      y[row] += sum;
    }
}
SpMV Performance I

GTX480/Nehalem (Core i7 950 @ 3.07 GHz)

CSR(scalar) vs MKL

CPU runs use all 4 cores

CSR-scalar problem: strided access leads to uncoalesced global memory load
Coalesced access

- If each work-item in a warp accesses the same segment, only one memory transaction.
- Otherwise, the number of transactions = number of segments accessed.
- Strided access leads to more than one segment accessed.
## Test Matrices Characteristics

<table>
<thead>
<tr>
<th></th>
<th>Dimension</th>
<th>Nonzeros</th>
<th>Nonzeros/Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cantilever</td>
<td>62K</td>
<td>4M</td>
<td>65</td>
</tr>
<tr>
<td>Circuit</td>
<td>171K</td>
<td>959K</td>
<td>6</td>
</tr>
<tr>
<td>Economics</td>
<td>207K</td>
<td>1.27M</td>
<td>6</td>
</tr>
<tr>
<td>Epidemiology</td>
<td>526K</td>
<td>2.1M</td>
<td>4</td>
</tr>
<tr>
<td>FEM/Ship</td>
<td>141K</td>
<td>3.98M</td>
<td>28</td>
</tr>
<tr>
<td>FEM/Sphere</td>
<td>83K</td>
<td>6M</td>
<td>72</td>
</tr>
<tr>
<td>Protein</td>
<td>36K</td>
<td>4.3M</td>
<td>119</td>
</tr>
<tr>
<td>Webbase</td>
<td>1M</td>
<td>3.1M</td>
<td>3</td>
</tr>
<tr>
<td>Wind Tunnel</td>
<td>218K</td>
<td>11.6M</td>
<td>53</td>
</tr>
</tbody>
</table>
Warp

- Workgroup divides into groups of 32 work-items, called warps
- Basic scheduling unit. All work-items within a warp always perform the same instruction
- When programming, one can think of the warp size as the SIMD vector length
__kernel void spmv_csr_vector {
    __local double buffer[BLOCK_SIZE];
    int tid = get_local_id(0);
    int gid = get_global_id(0);
    int wid = gid / WARP_SIZE; // warp id
    int lane = gid & (WARP_SIZE - 1); // local id within a warp
    int row = wid;

    if (row < nRows) {
        int start = rows[row];
        int end = rows[row+1];
        buffer[tid] = 0;
        for (int j = start+lane; j < end; j += WARP_SIZE) {
            buffer[tid] += vals[j]*x[cols[j]];
        }
        if (lane < 16) buffer[tid] += buffer[tid+16];
        if (lane < 8 ) buffer[tid] += buffer[tid+8 ];
        if (lane < 4 ) buffer[tid] += buffer[tid+4 ];
        if (lane < 2 ) buffer[tid] += buffer[tid+2 ];
        if (lane < 1 ) buffer[tid] += buffer[tid+1 ];
        if (lane == 0)
            y[row] += buffer[tid];
    }
}

One warp per row:

```
Row 0
Row 1
vals
  ...  ...
```

Every threads in the warp accumulate a partial sum into local memory if row size > WARP_SIZE

Performance a reduction from partial sum

Why written in this way?
Local Memory Bank Conflict

• Local memory is divided into banks
  – Successive 32-bit words assigned to successive banks
  – Number of banks: 32 in in Fermi, 16 in Tesla

• Bank conflict: two R/W fall in the same bank, the access will be serialized.

• Special cases
  – If all work-items in a warp access the same word, one broadcast, Fermi can also do multi-cast
  – If reading continuous byte, no conflict on Fermi
  – If reading double, no conflict on Fermi
Avoiding Bank-conflict in Local Memory Reduction

if (lane < 16) buffer[tid] += buffer[tid+16];
if (lane < 8  ) buffer[tid] += buffer[tid+8  ];
if (lane < 4  ) buffer[tid] += buffer[tid+4  ];
if (lane < 2  ) buffer[tid] += buffer[tid+2  ];
if (lane < 1  ) buffer[tid] += buffer[tid+1  ];
SpMV Performance II

GTX480/Nehalem

- Achieve better performance when nonzeros/row >= warp size
- CSR-vector Problem: wasted work-item when nonzeros/row < warp size
## Test Matrices Characteristics

<table>
<thead>
<tr>
<th></th>
<th>Dimension</th>
<th>Nonzeros</th>
<th>Nonzeros/Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cantilever</td>
<td>62K</td>
<td>4M</td>
<td>65</td>
</tr>
<tr>
<td>Circuit</td>
<td>171K</td>
<td>959K</td>
<td>6</td>
</tr>
<tr>
<td>Economics</td>
<td>207K</td>
<td>1.27M</td>
<td>6</td>
</tr>
<tr>
<td>Epidemiology</td>
<td>526K</td>
<td>2.1M</td>
<td>4</td>
</tr>
<tr>
<td>FEM/Ship</td>
<td>141K</td>
<td>3.98M</td>
<td>28</td>
</tr>
<tr>
<td>FEM/Sphere</td>
<td>83K</td>
<td>6M</td>
<td>72</td>
</tr>
<tr>
<td>Protein</td>
<td>36K</td>
<td>4.3M</td>
<td>119</td>
</tr>
<tr>
<td>Webbase</td>
<td>1M</td>
<td>3.1M</td>
<td>3</td>
</tr>
<tr>
<td>Wind Tunnel</td>
<td>218K</td>
<td>11.6M</td>
<td>53</td>
</tr>
</tbody>
</table>

Large nonzeros per row
COO Kernel

The kernel performance will be much less sensitive to sparsity pattern compared to CSR-vector
SpMV Performance III

GTX480/Nehalem

• More stable perf compared to CSR-Vector
• COO problem: lower compute-to-memory ratio. Load 4 data (col, indx, vals, x) per element calculation.
```c
__kernel void spmv_ell(__global int* inds,
                   __global double* vals,
                   __global double* x,
                   __global double* y,
                   int stride,
                   int nCols,
                   int nRows)
{
    int gid = get_global_id(0);
    if (gid < nRows) {
        double sum = 0.0;
        for(int i = 0; i < nCols; i++) {
            int idx = (i*stride + gid);
            int j = indsELL[idx];
            if (j != -1)
                sum += valsELL[idx] * x[j];
        }
        y[gid] += sum;
    }
}
```
In almost all cases, HYB is the highest performing kernel
# Test Matrices Characteristics

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimension</th>
<th>Nonzeros</th>
<th>Nonzeros/Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cantilever</td>
<td>62K</td>
<td>4M</td>
<td>65</td>
</tr>
<tr>
<td>Circuit</td>
<td>171K</td>
<td>959K</td>
<td>6</td>
</tr>
<tr>
<td>Economics</td>
<td>207K</td>
<td>1.27M</td>
<td>6</td>
</tr>
<tr>
<td>Epidemiology</td>
<td>526K</td>
<td>2.1M</td>
<td>4</td>
</tr>
<tr>
<td>FEM/Ship</td>
<td>141K</td>
<td>3.98M</td>
<td>28</td>
</tr>
<tr>
<td>FEM/Sphere</td>
<td>83K</td>
<td>6M</td>
<td>72</td>
</tr>
<tr>
<td>Protein</td>
<td>36K</td>
<td>4.3M</td>
<td>119</td>
</tr>
<tr>
<td>Webbase</td>
<td>1M</td>
<td>3.1M</td>
<td>3</td>
</tr>
<tr>
<td>Wind Tunnel</td>
<td>218K</td>
<td>11.6M</td>
<td>53</td>
</tr>
</tbody>
</table>

Very large nonzeros per row
Linear Solvers on GPU

• With SpMV kernels, one has the main ingredient for linear solvers
• The time to transform a CSR matrix to HYB and copy to GPU is $O(10-100)$ of a single SpMV call on GPU.
• If the solver involves $O(100-1000)$ iterations, use HYB. The expensive GPU matrix format transformation time for HYB (done on CPU) can be amortized.
• If the solver involves $O(10)$ iterations, CSR/COO may be better suited.
Pushing the boundary for HYB

• Overlap initial matrix format conversion and CPU-GPU data transfer using double buffering

• Matrix format conversion is easy to parallelize using, e.g. OpenMP/OpenCL on CPU
Summary

• GPU can delivers high SpMV performance for unstructured matrices
  – 10-20 Gflop/s
• In many cases, HYB (ELL+COO) format is the highest perf kernel.
• COO kernel perf shows least variations.
• Explore for your particular case!
• For further info on SpMV on GPU, see Bell & Garland’s paper in SC09
Questions

Contact:
penwang @nvidia.com
ibuck @nvidia.com
Agenda

• Review of basic constructs in OpenCL, 20 min
  – A quick revision of core OpenCL constructs
  – The future of OpenCL

• Examples (Part 1):
  – Conjugate gradient solver: sparse-matrix vector multiply kernel, 30 min
  – Device Fission and vector containers, 30 min

• Examples (Part 2):
  – N-body kernels, 30 min
  – FFT and wavelets, 30 min
  – Bilateral Gaussian, 15 – 30 min (depending on time)
Device Fission

Benedict R. Gaster, AMD
OpenCL™ extensions

• KRH (e.g. cl_khr_d3d10_sharing)
  – Developed and approved by the OpenCL™ working group
    • Included with OpenCL™ 1.1 conformance submission
      – Documentation on the Khronos OpenCL™ registry

• EXT (e.g. cl_ext_device_fission)
  – Developed by at 2 or more members of the working group
  – No required conformance tests
  – Documentation shared on the Khronos OpenCL™ registry

• Vendor (e.g. cl_amd_printf)
  – Documentation still shared on the Khronos OpenCL™ registry
OpenCL™ Device Fission (cl_khr_device_fission)

• Provides an interface for sub-dividing an OpenCL™ device into multiple sub-devices
• Typically used to:
  – Reserve a part of the device for use for high-priority/latency-sensitive tasks for the
  – Subdivide compute devices along some shared hardware feature like a cache

• Supported by CPU and Cell Broadband devices
  – Multicore CPU devices (AMD and Intel)
  – IBM Cell Broadband

• In the future may we see support also for the GPU?

• Developer by
  – AMD, Apple, IBM, and Intel
Parallel algorithms, containers, and tasks with OpenCL™

- Parallel algorithms
  - `parallel_for`
  - `parallel_reduction`
  - Asynchronous tasks with results (i.e. futures)

- Parallel containers
  - `vector` (including `push_back` and `insert`)
  - `queue` (blocking and unblocking)
  - `dictionary`

- For this talk we focus only on implementing `vector.push_back` on OpenCL™ CPU
High-Level containers and parallel tasking with OpenCL™

OpenCL runtime
Tasking & Futures

Append 0x00000000 after all 0x000000FF

parallel_for(A.begin(), x != A.end(), (T v) => {
  B.push_back(v);
  if (v == 0x000000FF) {
    B.push_back(0x00000000);
  }
});

Write to temp
Prefix sum
Copy to final

Append serially

Reference:
SPAP: A Programming Language for Heterogeneous Many-Core Systems, Qiming Hou et al, 2010.
High-Level containers and parallel tasking with OpenCL™

```cpp
parallel_for(A.begin(), x != A.end(), (T v) => {
    B.push_back(v);
    if (v == 0x000000FF) {
        B.push_back(0x00000000);
    }
});
```

- Write to temp
- Prefix sum
- Copy to final
- Append in Parallel
- Outlined in this talk

Append 0x00000000 after all 0x000000FF

OpenCL runtime

Tasking & Futures
Containers have sequential semantics

```c
unsigned int padding(unsigned int n, unsigned int in[n],
    unsigned int out[n*2])
{
    for (int i = 0, j = 0; i < n; i++, j++) {
        unsigned int x = in[i];
        out[j] = x;
        if (x == 0xFF) {
            out[++j] = 0x0;
        }
    }
}
```

- Focus on `push_back()` implementation
- Simple C implementation.
Containers have sequential semantics

```c
unsigned int padding(unsigned int n, unsigned int in[n],
                      unsigned int out[n*2])
{
    for (int i = 0, j = 0; i < n; i++, j++) {
        unsigned int x = in[i];
        out[j] = x;
        if (x == 0xFF) {
            out[++j] = 0x0;
        }
    }
}
```
Naïve approach might simply replace the for with parallel_for

```c
__kernel void padding(
    __global uint * counter,
    __global uint * input,
    __global uint * output)
{
    uint x = input[get_global_id(0)];
    uint offset;
    if (x == 0xFF) {
        offset = atomic_add(counter, 2);
        output[offset] = x;
        output[offset+1] = 0x0;
    }
    else {
        offset = atomic_inc(counter);
        output[offset] = x;
    }
}
```
Naïve approach might simply replace the for with parallel_for

```c
__kernel void padding(
    __global uint * counter,
    __global uint * input,
    __global uint * output)
{
    uint x = input[get_global_id(0)];
    uint offset;
    if (x == 0xFF) {
        offset = atomic_add(counter, 2);
        output[offset]   = x;
        output[offset+1] = 0x0;
    } else {
        offset = atomic_inc(counter);
        output[offset]   = x;
    }
}
```

Problem:

Padding values are correctly inserted but there is no guarantee that original ordering will be preserved!
Naïve approach might simply replace the for with parallel_for

```c
__kernel void padding(
    __global uint * counter,
    __global uint * input,
    __global uint * output)
{
    uint x = input[get_global_id(0)];
    uint offset;
    if (x == 0xFF) {
        offset = atomic_add(counter, 2);
        output[offset] = x;
        output[offset+1] = 0x0;
    }
    else {
        offset = atomic_inc(counter);
        output[offset] = x;
    }
}
```

Problem:

Padding values are correctly inserted but there is no guarantee that original ordering will be preserved.
Recast as parallel pipeline pattern

1. **read and compute block into local memory**
2. **write offset**
3. **output padded block**

4. **mailbox**
5. **read offset**
6. **write offset**
7. **output padded block**

8. **read and compute block into local memory**
9. **mailbox**
10. **read offset**
11. **write offset**
12. **output padded block**

13. **Read offset**
14. **Write offset**
15. **output padded block**

16. **mailbox**
Pipeline 1

__kernel void padding( __global uint * counter, __global uint * groupid,
                    __global uint * input, __global uint * output)
{
    uint x   = input[get_global_id(0)];
    size_t lid = get_local_id(0);
    size_t gid = get_group_id(0);
    if (gid != 0) {
        if (lid == get_local_size(0) - 1) {
            while(1) {
                if (*groupid == (gid -1)) {
                    break;
                }
            }
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
Pipeline 1\textsuperscript{st} attempt cond

uint offset;
if (x == 0xFF) {
    offset = \texttt{atomic\_add}(counter, 2);
    output[offset] = x;
    output[offset+1] = 0x0;
}
else {
    offset = \texttt{atomic\_inc}(counter);
    output[offset] = x;
}

if (lid == \texttt{get\_local\_size}(0) - 1) {
    *\texttt{groupid} = gid;
}
}
The problem with this approach

OpenCL™ makes no guarantee about work-group execution order

Problem:

Native approach has no guarantee of progress

Note, this is not resolved by issuing only two work-groups
Device Fission can guarantee progress

Solution:

Multiple devices guarantee that each will make progress independent of the other.

Each device must have a corresponding command queue.

Launch one work-group per command queue (often referred to as fill the machine).

Use native kernels to manually control “work-group” configuration.
Native Kernels

• Enqueue C/C++ functions, compiled by the host compiler, to execute from within an OpenCL™ command queue

```c
cl_int clEnqueueNativeKernel (cl_command_queue command_queue,
    void (*user_func)(void *),
    void *args,
    size_t cb_args,
    cl_uint num_mem_objects,
    const cl_mem *mem_list,
    const void **args_mem_loc,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

• There is no guarantee that the function will execute in same thread that the enqueue was performed; must be careful about thread-local-storage usage
Pipeline 2\textsuperscript{nd} attempt (host device fission code)

```cpp
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
if (platforms.size() == 0) {
    // handle error case
}
cl_context_properties properties[] = {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platforms[0])(), 0};
cl::Context Context(CL_DEVICE_TYPE_CPU, properties);
std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
if (devices[0].getInfo<CL_DEVICE_EXTENSIONS>().find("cl_ext_device_fission")
    == std::string::npos) {
    // handle case when fission not supported, e.g. fall back to sequential version
}
Pipeline 2\textsuperscript{nd} attempt (host device fission code)

```cpp
cl_device_partition_property_ext subDeviceProperties[] =
    { CL_DEVICE_PARTITION_EQUALLY_EXT, 1,
      CL_PROPERTIES_LIST_END_EXT, 0};

devices[0].createSubDevices(subDeviceProperties, &subDevices);
if (subDevices.size() <= 0) {
    // handle error case
}

counterV = new cl_uint[subDevices.size()]; // mailboxes

unsigned int j = 0;
for (std::vector<cl::Device>::iterator i = subDevices.begin();
    i != subDevices.end(); i++) {
    queues.push_back(cl::CommandQueue(context, *i)); counterV[j++] = 0;
}

// Code to allocate and setup buffers and so on
```
Pipeline 2\textsuperscript{nd} attempt (host device fission code) cond

```cpp
cl_uint * args[6] = { NULL, NULL, NULL, NULL,
                      reinterpret_cast<cl_uint *>(size/numComputeUnits), 0x0 }; 

std::vector<cl::Memory> memArgs;
memArgs.push_back(input);   memArgs.push_back(output);
memArgs.push_back(counter); memArgs.push_back(blocks[0]);

std::vector<const void *> memLocations;
memLocations.push_back(&args[0]); memLocations.push_back(&args[1]);
memLocations.push_back(&args[2]); memLocations.push_back(&args[3]);

unsigned int groupID = 0;
std::vector<cl::Event> events;
```
Pipeline 2\textsuperscript{nd} attempt (host device fission code) cond

groupID = 0;
cl::Event event;
for (unsigned int i = 0; i < numComputeUnits; i++) {
    memArgs.pop_back(); memArgs.push_back(blocks[i]);
    args[5] = reinterpret_cast<cl_uint*>(groupID);
    queues[i].enqueueNativeKernel(
        padding,
        std::make_pair(static_cast<void*>(args),
                        sizeof(cl_uint*)*arraySize(args)),
        &memArgs, &memLocations, NULL, &event);
    events.push_back(event);
    groupID++;
}
cl::Event::waitForEvents(events);
Pipeline 2\textsuperscript{nd} attempt (native function)

```c
void padding(void * args)
{
    unsigned int ** argsPtr = static_cast<unsigned int **>(args);
    unsigned int * input = *argsPtr++;
    unsigned int * output = *argsPtr++;
    volatile unsigned int * counter = *argsPtr++;
    unsigned int * localBlock = *argsPtr++;

    unsigned int blockSize = reinterpret_cast<unsigned int>((*argsPtr++));
    unsigned int groupId = reinterpret_cast<unsigned int>(*argsPtr);

    unsigned int offset = groupId * blockSize;
    unsigned int localBlockSize = 0;
}
```
Pipeline 2\textsuperscript{nd} attempt (native function) cond

for (unsigned int i = offset; i < blockSize+offset; i++, localBlockSize++) {
    unsigned int x = input[i];
    localBlock[localBlockSize] = x;
    if (x == 0x000000FF)
        localBlock[++localBlockSize] = 0x0;
}

if (groupId > 0) {
    offset = counter[groupID-1];
    while (offset == 0)
        offset = counter[groupID-1]; // read mailbox
}
counter[groupID] = localBlockSize+offset; // write to mailbox

for (unsigned int i = offset, j = 0; i < localBlockSize+offset; i++, j++)
    output[i] = localBlock[j];
A lot more can still be done...

- Presented implementation is not optimized for CPU cache usage
  - Divide work into L1-D cache size blocks of work
  - Extend mailbox scheme to support multiple iterations, so
    - Core 1 – $B_0, \ldots, B_{\#cores}, \ldots, B_{\#cores\times2}$
    - Core 2 – $B_{1}, \ldots, B_{\#cores+1}, \ldots, B_{(\#cores\times2)+1}$
    - ...
- Presented implementation does not use blocking mailboxes
  - Statically sized queues
    - Block when writing to a full queue (not needed for this algorithm)
    - Block when reading from an empty queue
- Avoid using native kernels (or even device fission)?
  - Directly through support for global work-group synchronization
  - OpenCL™ statically sized queues (as above), that use global work-group synchronization underneath
Questions

Contact:

benedict.gaster@amd.com
Agenda

• Review of basic constructs in OpenCL, 20 min
  – A quick revision of core OpenCL constructs
  – The future of OpenCL

• Examples (Part 1):
  – Conjugate gradient solver: sparse-matrix vector multiply kernel, 30 min
  – Device Fission and vector containers, 30 min

• Examples (Part 2):
  – N-body kernels, 30 min
  – FFT and wavelets, 30 min
  – Bilateral Gaussian, 15 – 30 min (depending on time)
N-Body Simulation

• Numerically Simulate evolution of system of N bodies
  – Each body continuously interacts with all other bodies

• Examples:
  – Astronomical and astrophysical simulation
  – Molecular dynamics simulation
  – Fluid dynamics simulation
  – Radiometric transfer (Radiosity)
  – … and many more

• Directly computing all-pair interactions results in $O(N^2)$ interactions to compute per time step
All-Pairs N-body Simulation

Given $N$ bodies with an initial position $x_i$ and velocity $v_i$ for, the force $f_{ij}$ on body $i$ caused by body $j$ is given by following (for the case of a gravitational potential):

$$f_{ij} = G \frac{m_i m_j}{\|r_{ij}\|^2} \cdot \frac{r_{ij}}{\|r_{ij}\|}, \quad F_i = \sum_{1 \leq j \leq N, j \neq i} f_{ij}$$

where $m_i$ and $m_j$ are the masses of bodies $i$ and $j$, respectively; $r_{ij} = x_j - x_i$

The acceleration is computed as $a_i = F_i / m_i$
Reducing terms in the force sum

- Consider computing force on earth due to all celestial bodies
  - Look at night sky, # terms in force sum >= number of visible stars
  - Oops! One “star” is really the Andromeda galaxy, which contains billions of real stars
    - Seems like a lot more work than we thought ...
- Don’t worry, ok to approximate all stars in Andromeda by a single point at its center of mass (CM) with same total mass
  - D = size of box containing Andromeda , r = distance of CM to Earth
  - Require that D/r be “small enough”

From UCB CS267, 2007
Apply recursively

- From Andromeda’s point of view, Milky Way is also a point mass
- Within Andromeda, picture repeats itself
  - As long as $D_1/r_1$ is small enough, stars inside smaller box can be replaced by their CM to compute the force on Vulcan
  - Boxes nest in boxes recursively

From UCB CS267, 2007
N body methods for long range forces

• Apply recursively to cut cost to \(O(n \log n)\):
  – Barnes-Hut algorithm

• For some special forms (e.g. \(1/r\)) we can use more sophisticated representations of distant interactions to reduce cost to \(O(n)\):
  – Fast Multipole method (FMM)

• But the “big O” analysis hides the complexity of the “multiplying constant”.
  – For modest sized \(N\), the direct “all-pairs” method is competitive.
  – And even in a recursive decomposition, you still need to compute interactions “inside a box”.

• Hence there is both pragmatic and pedagogical value to looking at the direct, all-pairs method.
N-body: Mathematical details

20 Flops per iteration:
- 3 Flops for computation of distance “r” in x, y, z directions (3 SUB)
- 6 Flops for computation of cumulative distance square + \( \varepsilon^2 \) (3 MUL, 3 ADD)
- 4 Flops for inverse distance cube (2 MUL, 1 SQRT, 1 INV)
- 1 Flops for multiplying mass with the inverse distance cube (1 MUL)
- 6 Flops for acceleration calculation in x, y, z directions (3 MUL, 3 ADD)

\[ F_i \approx G m_i \cdot \sum_{1 \leq j \leq N} \frac{m_j r_{ij}}{\left( ||r_{ij}||^2 + \varepsilon^2 \right)^{3/2}}. \]

Softening factor ... limits force between pairs to stabilize later integrations
Explicit SIMD data parallelism:
- The kernel defines one stream of instructions
- Parallelism from using wide vector types
- Size vector types to match native HW width
- Combine with task parallelism to exploit multiple cores.

Implicit SIMD data parallelism (i.e. shader-style):
- Write the kernel as a “scalar program”
- Kernel automatically mapped to SIMD-compute-resources and cores by the compiler/runtime/hardware.

Both approaches are viable options ... the implicit method leads to code with more “portable performance” ... the explicit method supports tuning for ultimate performance
Performance Results

**NBody Performance**

Results from Intel’s internal OpenCL implementation:

- Porting C to CL - Implicit Data Parallelism
  - “shader-style” code
  - Benefit from multi-core/SMT

- Implicit Data Parallelism with Vectorization
  - Intel’s internal implementation
  - Cross-workitem Vectorization/Packing
  - Benefit from SSE (128bit registers)

- Explicit Data-Parallelism
  - Hand tuned OpenCL C code

* Results measured on Core™ i7 975, 3.3 GHz, 6GB DDR3
* Results depends on the algorithm/code running

From Ofer Rosenberg’s presentation at 2010 SIGGRAPH OpenCL BOF
N body Methods in OpenCL

• We will explore OpenCL for N body methods in more detail:
  – Demos to show that it works
  – Code walk throughs ... so you can see the impact of optimizations “in real time”
Questions

Contact:

timothy.g.mattson @intel.com
Agenda

• Review of basic constructs in OpenCL, 20 min
  – A quick revision of core OpenCL constructs
  – The future of OpenCL

• Examples (Part 1):
  – Conjugate gradient solver: sparse-matrix vector multiply kernel, 30 min
  – Device Fission and vector containers, 30 min

• Examples (Part 2):
  – N-body kernels, 30 min
  – FFT and wavelets, 30 min
  – Bilateral Gaussian, 15 – 30 min (depending on time)
Discrete Fourier Transforms

Mike Houston

AMD
**Background**

• Lots of work on FFT algorithms
  – Detailed survey in [Van Loan 92]

• Lots of work on FFTs on GPUs
  – See “High Performance Discrete Fourier Transforms on GPUs” [Govindaraju SC08]
  – CUFFT
  – Microsoft DFT library
  – Apple OpenCL FFT library

• I’m going to cover some of the basics of what to think about for tuning
Discrete Fourier Transforms (DFTs)

• Converts data in the spatial or temporal domain into frequencies the data comprises.

\[
F\{f(x)\} = F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x)W_N^{ux}
\]

\[
F^{-1}\{F(u)\} = f(x) = \sum_{u=0}^{N-1} F(u)W_N^{-ux}
\]

– Often computed using Fast Fourier Transforms (FFTs) for efficiency
  – This talk will concentrate on FFTs

• Fundamental primitive for signal processing
  – Convolutions, compression, classification, cryptography, computational fluid dynamics, large polynomial multiplications, image and audio processing, etc.
Multi-Dimensional DFTs

- 2D transform can be computed by applying the transform in one direction, then the other.

$$F\{f(x, y)\} = F(u, v) = \frac{1}{MN} \sum_{y=0}^{N-1} \sum_{x=0}^{M-1} f(x, y)W_M^{ux}W_N^{vy}$$

$$F^{-1}\{F(u, v)\} = f(x, y) = \sum_{v=0}^{N-1} \sum_{u=0}^{M-1} F(u, v)W_M^{-ux}W_N^{-vy}$$

Courtesy Kenneth Moreland
General Approach

• An FFT of length N is usually carried out as a series of passes:
  \[ V_1 = \text{Pass}_1(V_0) \]
  \[ V_2 = \text{Pass}_2(V_1) \]
  \[ \vdots \]
  \[ V_P = \text{Pass}_P(V_{prev}) \]

• where each pass divides the N elements into N/R vectors of length R, applies a length R DFT to each piece, multiplies each point by a unit-length complex “twiddle factor” and then reassembles the length N vector.

• The number of passes depends on the value of each R, and
  \[ N = R_1 \, R_2 \, R_3 \, \ldots \, R_P \]
  \[ 1024 = 2^{10} = 4^5 = 32^2 = 16 \times 4 \times 16 = 8 \times 4 \times 4 \times 2 \times 2 \times 2 \]
DFT: Challenges

• Low arithmetic intensity

• Complex memory access patterns
  – Limited data reuse
  – *It’s all about the memory system utilization!*

• Architectural issues
  – Cache associativity, memory banking/interleaving, data access latency, available device memory
  – On GPU, resource utilization impacts work-items in flight, which impacts latency hiding and efficiency
Many Tuning Options

• Use local memory?
• What R’s to use?
• Which order of R’s to use?
• Which work group size?
• Whence the twiddle factors?
• How much local memory?
• Memory size vs. memory access pattern?
• How many registers?
• Whence the data?
Option 1: use local memory

• Without local memory each pass will require a round trip of data from global memory. Also element access during the pass may not hit RAM efficiently (due to complicated addressing), although coalesce hardware may help.

• If local memory is used, some fraction (usually ½ or 1) of the data to be transformed will be kept in local memory, and transfers to and from global memory will only be needed on the first and last passes. However, local memory use can restrict the number of work items in flight.
Option 2: What R’s to use

- For a complex FFT, a given length R DFT will require around $2R$ registers.
- Near-optimal operation count short DFTs may require several more than $2R$ registers depending on the compiler.
- Given a focus on memory system, it makes sense to look for minimal register implementations.
- Larger R means fewer passes but also potentially fewer work items in flight.
  - Less items in flight means less latency hiding
- Depending on the work group size, a given work item may need to process more than one of the $N/R$ DFT’s at the expense of more registers and therefore possible reduction of work items in flight.
Option 3: Which order of R’s to use

• The order of the R’s affects indexing into the vector to perform the N/R length R DFTs, and which and how many twiddle factors are needed.

• While it is possible to model the effects of different orders in some cases, an implementation that can dynamically determine the best order for the specific device is advisable.
Option 4: Which workgroup size?

- A given OpenCL FFT kernel can adapt itself to the work group size it is given at the cost is significant logic and indexing complexity.
- However, control flow can be completely eliminated and indexing computation greatly reduced by using a kernel targeted for a specific N and work group size.
  - Online recompilation via OpenCL compiler really useful here...
- The most efficient work group sizes are likely to be multiples of the native hardware execution width
  - wavefront size in AMD speak/warp size in Nvidia speak
  - Query device for `cl_kernel_preferred_work_group_size_multiple`
Option 4, continued

- Given work group size $G$, $N$, and $R$, to balance the work, $N/R$ must be divisible by $G$.
  - Also remember that NDRange must be divisible by $G$
- The value $N/R/G$ is the number of length $R$ DFTs each work item will carry out in a given Pass.
  - Note that $N/R$ divisible by $G$ is an additional constraint on Option 2.
- For example, for $N=1024$, and $G=64$, $R$ must be 2, 4, 8, or 16.
- If $N/R$ divisible by $G$ is not possible or convenient (e.g. when $N$ is small), the kernel can be designed to simultaneously handle a batch of $B$ transforms. Then $BN/R$ must be divisible by $G$. 
Option 5: Whence the twiddle factors?

• The twiddle factors used in FFT are primitive Nth roots of unity (or simple functions thereof) and are of the form \( \cos(A) + i \sin(A) \) (where “\( i \)” is the imaginary value \( \sqrt{-1} \), and \( A = \frac{2 \pi K}{N} \), where \( 0 \leq K < N \).

• There are a variety of options for obtaining the twiddle factors affecting both performance and accuracy.
  – Computed directly using the OpenCL built in sin and cos functions.
  – Computed directly using half_sin and half_cos functions
  – Computed using native_sin and native_cos functions.

  • GPUs typically have limited accuracy machine instructions for sin and cos that are exposed by the native_sin and native_cos built in functions. They can offer the highest performance, but the cost is a reduced accuracy result. But, some native implementation are accurate “enough” for some FFTs
Option 5, continued

• Other options are to save the twiddle factors in:
  – constant buffer
    • __constant array in the program
  – image (texture)
  – global buffer.
  – The first two are likely to be faster

• A combination of approaches of varying accuracy may be used based on various instances of the “double angle” formulas of trigonometry.
Option 6: How much local memory?

- A given work item may require data produced by a different work item on the previous pass. Local memory is needed to pass data between work items in a workgroup without going to global memory.
- The trade-off is local memory size vs. complexity and number of barriers.

- Simplest case:
  - Compute size R FFT(s)
  - Save entire vector to local memory
  - Barrier
Option 6, continued

• If the entire vector is able to be held in registers, we can halve the amount of local memory by

  Compute size R DFT(s)
  Save real part of vector to local memory
  Barrier
  Read real part of vector from local memory
  Barrier
  Save imaginary part of vector to local memory
  Barrier
  Read imaginary part of vector from local memory
Option 7: Memory size vs. access pattern

• Local memory
  – Local memories are often banked structures and performance may be reduced when the work group only accesses a small number of banks on a given access. ("bank conflicts")
  – There are a number of techniques to reduce conflicts including more complicated addressing and/or padding. Increasing the size of the local memory can negatively impact performance, so the benefit of padding must be measured.
  – Affine access patterns can map to same banks
Option 7: Memory size vs. access pattern

• Global memory
  – “polite” memory access patterns work better through caches and wide memory interfaces, especially those on GPUs
    • Minimum read burst on GPUs 256+ bytes
  – May want to explore memory layout, and padding
  – **Access contiguous blocks of data to achieve high DRAM bandwidth**

• Watch out for the cost of transposing data
Option 8: How many registers?

• Recall option 2:
  – “For a complex FFT, a given length \( R \) DFT will require around \( 2R \) registers. Near-optimal operation count short DFTs may require several more than \( 2R \) registers depending on the compiler.”
• The number of registers used affects the number of work items in flight, and therefore the hiding of global memory latency.
• While the programmer has no direct control of this, unless you are doing this in assembly, there is the ability to influence it.
Option 9: Whence the data?

• Up to this point, it has been assumed data will be read from or written to either local or global memory.
  – OpenCL provides another option: *images*.

• Images provide a way to use texture hardware and caching on GPUs
  – GPU caches are generally 2D, so keep that in mind
  – Are caches large enough to absorb the vectors being transformed?

• Some GPUs have read/write caches, but using images may still be faster
FFT Overview
FFT Overview

FFT along columns

Transpose

FFT along rows

Courtesy Naga Govindaraju
Registers

Global memory (>=256MB)

Local Memory (16-32KB/compute-unit)

Global memory (>=256MB)

Courtesy Naga Govindaraju
Global memory algorithm

• Proceeds in $\log_R N$ steps (radix=$R$)
• Decompose $N$ into work groups $G$, and work-items $W$ such that $G \times W = N/R$
• Each work-item:
  – reads $R$ values from global memory
  – multiplies by twiddle factors
  – performs an $R$-point FFT
  – writes $R$ values back to global memory
Global Memory Algorithm

If \( N/R > \text{coalesce width (CW)} \), no coalescing issues during reads

If \( R^j > CW \), no coalescing issues during writes

If \( R^j \leq CW \), write to local memory, rearrange data across work-items, write to global memory with coalescing

Courtesy Naga Govindaraju
Local memory algorithm

• Applied when FFT is computed on data in local memory of a compute unit
• Each workgroup has N*M/R work-items
  – M is number of FFTs performed together in a workgroup
  – Each compute unit performs M FFTs at a time
• Similar to global memory algorithm
  – Possibly use Stockham formulation to reduce compute overheads
Local Memory Algorithm

If N/R > numbanks, no bank conflicts during reads

If R^j > numbanks, no bank conflicts during writes

Courtesy Naga Govindaraju
Local Memory Algorithm

If $R^i \leq \text{numbanks}$, add padding to avoid bank conflicts

R=4
Step j=1

N/R

work-item 0
work-item 1
work-item 2
work-item 3
work-item 4
work-item 5
work-item 6
work-item 7

Banks

R^i

0 4 8 12 0 4 8 12

Courtesy Naga Govindaraju
Hierarchical FFT

- Decompose FFT into smaller-sized FFTs
  - Evaluate efficiently using local memory algorithm
  - Combine transposes with FFT computation
  - Match memory coalescing behavior of hardware
Other things to consider

• Non-Power-of-Two sizes
  – Mixed Radix
    • Using powers of 2, 3, 5, etc.
  – Bluestein’s FFT
    • For large prime factors

• Multi-dimensional FFTs
  – Perform FFTs independently along each dimension

• Real FFTs
  – Exploit symmetry to improve the performance
  – Transformed into a complex FFT problem

• DCTs
  – Computed using a transformation to complex FFT problem
Specific usage case: 1024 complex FFTs

- ATI Radeon 5870
  - Use local memory? Yes
  - What size R’s to use? 4 only
  - Which order of R’s to use? Doesn’t matter, \(1024 = 4^5\)
  - Which work group size? 64
  - Whence the twiddle factors? native_sin, native_cos
  - How much local memory? 4352 bytes
  - Local memory size vs. conflict avoidance? Padding.
  - How many registers? Absorb entire vector into registers. Optimal operation count length 4 DFT.
  - Whence the data? Global buffers.

- “5 N \log_2(N)” flop count for FFT = 391Gflop/s (kernel only time)
- Assuming infinitely fast computation and global memory bandwidth of 153GB/s, the (unobtainable) limit for this transform on this device is 478Gflop/s.
Conclusions

• Fast arbitrarily sized FFTs tricky
  – Many variants
  – Lots of different architectures

• How to build a library of these?
  – OpenCL device queries
    • Get hardware information
  – Specialize kernels for data size
    • Complete auto-tuned libraries?
Acknowledgments

• Naga K. Govindaraju, Microsoft
  – See Naga’s excellent SC08 paper and presentation on a CUDA based FFT library that goes quite deep on building FFT software.
• Brian Sumner and Jim Conyngham, AMD
Questions

Contact:

Michael.Houston@AMD.com
Brian.Sumner@AMD.com
Agenda

• Review of basic constructs in OpenCL, 20 min
  – A quick revision of core OpenCL constructs
  – The future of OpenCL

• Examples (Part 1):
  – Conjugate gradient solver: sparse-matrix vector multiply kernel, 30 min
  – Device Fission and vector containers, 30 min

• Examples (Part 2):
  – N-body kernels, 30 min
  – FFT and wavelets, 30 min
  – Bilateral Gaussian, 15 – 30 min (depending on time)
Bilateral Gaussian Blur

Benedict R. Gaster
Saif Ali

AMD
Gaussian Blur

- Image convolution – convolve image with 2D Gaussian filter
- Parallelizes trivially – independent computation at each pixel
- Separable kernel – optimized as two 1D passes
- Data independent – same kernel applied everywhere, destroys edges
Bilateral Gaussian Blur

- Edge preserving blur
- Non-separable kernel – cannot implement as two passes
- Data dependent - Shape of kernel depends on image content
Bilateral Gaussian Blur

- Use two Gaussian kernels
- One for spatial variation (as in traditional Gaussian blurring)
- A second one for variation in image values – range

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(|p-q|) G_{\sigma_r}(||I_p-I_q||)$$

- Smaller value when points p and q are distant in space
- Smaller value when points p and q are distant in range, like at edges
Bilateral Gaussian Blur

\[ BF[I]_p = \frac{1}{w_p} \sum_{q \in S} G_{\sigma_s}(|p-q|) G_{\sigma_r}(||I_p-I_q||) \]

Range – modulates the spatial blur based on image content

From Durand, 2002
Challenges

• Non-separable kernel - cannot implement as two 1-D passes
• Data dependent computation - kernel cannot be pre-computed
• Cannot be done in frequency domain - not strictly a convolution
GPU Computation – OpenCL

• Parallelizes easily as computation at each pixel is dependent only on local neighborhood
• Implemented with compute shaders
• Gaussian kernels are pre-computed and stored in a texture
• Bilateral Gaussian kernel computed on-the-fly
// OpenCL Bilateral Gaussian Kernel
__kernel void bilateralGaussianBlur(
__global  float * output,
__global  float * input,
__global  float * gaussianBuffer,
int kernelSize,
float sigma_d,
float sigma_r )

- **output** – The blurred image
- **Input** – The input image
- **gaussianBuffer** – Pre-computed Gaussian kernels for the space component
- **kernelSize** – The size of the filtering kernel
- **sigma_d** – The standard deviation for the space Gaussian
- **Sigma_r** – The standard deviation for the range Gaussian
GPU Computation - OpenCL

Divide input image into blocks
Assign each block to one work-group

Compute Domain
$n \times n$ blocks
$m \times m$ work items per block

$N \times N$ image
$n \times n$ blocks
$n = N/4$
GPU Computation - OpenCL

- At each pixel sample $k \times k$ neighborhood ($k = \text{kernelSize}$)
- Multiply space and range Gaussian functions to get final result

```c
// Bilateral Gaussian Blur
for(int i = -kernelSizeBy2; i <= kernelSizeBy2; i ++)
{
    for(int j = -kernelSizeBy2; j <= kernelSizeBy2; j ++)
    {
        // input image
        tempTid.x = clamp(globalId.x+i, 0, 511);
        tempTid.y = clamp(globalId.y+j, 0, 511);
        inAddr = addrTranslate2DTO1D( tempTid, 512, 4 )/4;
        Iq = input[ inAddr ];
        // space Gaussian, look up buffer with precomputed coefficients
        offset = (kernelSize/2) * (2*kernelSize/2 - 1) * (2*kernelSize/2 + 1) / 3;
        offset --;
        offset += ((i+kernelSizeBy2)*kernelSize + (j+kernelSizeBy2));
        space = gaussianBuffer[ offset*4 ];
        // range Gaussian - compute on the fly
        range = gaussian1D( Ip - Iq, 1.0f, sigma_r );
        // compute normalization factor
        Wp += (space * range);
        val += (space * range * Iq);
    }
}
```
Oversampling Problem

- At each pixel sample $k \times k$ neighborhood
- Neighborhoods overlap for adjacent pixels

Neighborhood Overlap for 3 adjacent pixels
Tiling Solution – Local Data Share

• Divide the input image into tiles
• Each tile is computed by one wavefront
• Load tile into local data share (LDS)
Bilateral Gaussian Blur in OpenCL

• If there is time we will expand on the design and implementation of bilateral gaussian blur using OpenCL:
  – Demos to show that it works
  – Code walk throughs ... so you can see the impact of optimizations “in real time”
Questions

Contact:

Benedict.Gaster@AMD.com
Saif.Ali@AMD.com