OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

W. Michael Brown
brownw@ornl.gov
Last update: 9/14/2012

Geryon

A Unified Coprocessor Library for OpenCL, CUDA-RT, and CUDA Driver

CUDA Runtime

- **One compiler interface for host and GPU code**
  - **Host and device code in a single file**
- **One executable for host and GPU code**
- **Succinct interface**
- **Currently the most used API for NVIDIA hardware**
- **Compared to OpenCL, can evolve more quickly to efficiently support NVIDIA hardware**

- Only supports NVIDIA hardware
- Need to maintain separate code for GPU and CPU calculations
  - Cannot run GPU kernels (*efficiently*) on CPU
- The compiler interface drives a host compiler and a GPU compiler
  - Very limited host compiler choice (GNU/Intel/Visual Studio)
  - Can be limited to certain host compiler versions
  - Must handle flags and error output from both compilers

CUDA Driver

•Freedom in choice and version of host compiler
•Some, but not much additional flexibility/functionality
•More similar to the OpenCL API
•Compared to OpenCL, can evolve more quickly to efficiently support NVIDIA hardware
•Supports JIT compilation of PTX code for different NVIDIA GPUs

•Only supports NVIDIA hardware
•Need to maintain separate code for GPU and CPU calculations
  •Cannot run GPU kernels (*efficiently*) on CPU
•Need to store and load GPU kernels in separate file(s) from executable or manage as strings from compiled output inside of code
•API is much more tedious than CUDA-RT

OpenCL

- **Supported by many vendors (not restricted to NVIDIA)**
- **Can run same kernels on multicore CPU and GPU**
- **Driver includes a compiler**
    - **Device kernels can be compiled on-the-fly for new architectures**
- **Freedom in choice of host compilers/versions**
- **Smart pointers (reference counting) are nice for library development**

- Less mature
- No templates, no pointer traversal
- No device pointer arithmetic on the host
- Need to store and load GPU kernels in separate file(s) from executable or manage kernels as strings for on-the-fly
- API is much more tedious than CUDA-RT
- Device kernels can be compiled on-the-fly for new architectures

OpenCL Differences (1)

for CUDA programmers

- OpenCL does not allow templates
    - Must be done the C-way with preprocessor directives
- OpenCL does not allow pointer traversal
- OpenCL does not allow device pointer arithmetic on the host
    - Must use integer offsets

```
CUDA:   gpu_sort(vector.begin(), vector.begin()+n);
OpenCL: gpu_sort_cl(vector.begin(),n);
```

OpenCL Differences (2)

for CUDA programmers

- OpenCL has no 2D memory copy routines for maintaining byte alignments
- In OpenCL, cannot bind existing array to 1D texture/image (to enable cache on certain NVIDIA devices)
- OpenCL has no memset routine for device memory
- OpenCL does not have 3-element vector types (*e.g. float3*)
  - Adding your own is discouraged for performance reasons
- OpenCL will not accept all types as kernel arguments
  - Some types (such as bool) that change byte size between compilers are not allowed
- OpenCL often requires qualifying device pointers with __global or __local to explicitly specify the address space

OpenCL Differences (3)

for CUDA programmers

- Many OpenCL routines require device/context/command_queue objects to be passed around
  - For CUDA, there is an implicit default command queue (stream) and most functions do not require device, context, and queue objects
- OpenCL uses reference counting pointers
- OpenCL allows for more advanced error handling

Geryon

- Intended to be a simple library for using CUDA Runtime, CUDA Driver, and OpenCL APIs with a consistent interface
  - Change from one API to another by simply changing the namespace
  - Use multiple APIs in the same code
  - Lightweight (only include files – no build required)
  - Manage device query and selection
  - Simple vector and matrix containers
  - Simple routines for data copy and type casting
  - Simple routines for data I/O
  - Simple classes for managing device timing
  - Simple classes for managing kernel compilation and execution

Geryon

Selecting a low-level API

```
#ifdef USE_CUDART_API
#include “nvc_device.h”       // For device routines
#include “nvc_mat.h”          // For vec/mat containers
#include “nvc_timer.h”        // For timing routines
using namespace ucl_cudart;
#endif

#ifdef USE_CUDADR_API
#include “nvd_device.h”
#include “nvd_mat.h”
#include “nvd_timer.h”
using namespace ucl_cudadr;
#endif

#ifdef USE_OPENCL_API
#include “ocl_device.h”
#include “ocl_mat.h”
#include “ocl_timer.h”
using namespace ucl_opencl;
#endif
```

Geryon

Device Management

```
// Each device object has information on the number of
// devices and device properties
// - See docs for full list of device property accessors
UCL_Device dev;
if (dev.num_devices()==0) exit(1);
cout << dev.name(0) << " has " << dev.gigabytes(0)
    << " of memory.\n";

// You must have one device object for each device you
// will be using. For CUDA each device object must be on
// a separate thread or process. The set() command will
// initialize the chosen device for use.
dev.set(0);

// The set command creates a default command queue that
// is stored in the device object (in CUDA this is the
// null stream)
command_queue &cq=dev.cq();

// You can add/delete additional command queues (streams)
dev.push_command_queue();
command_queue &cq_kernels=dev.cq(1);
dev.pop_command_queue();
```

```
// To block until all commands in the default queue
// (stream) have completed (synchronization)
dev.sync();


// To block until all commands in the second command
// queue associated with a device have completed:
dev.sync(1);
// -- or --
ucl_sync(dev.cq(1));

// To block until a certain command in the queue has
// completed use timer objects (see below)
```

Default Command Queue

- If a command queue is not specified, all data transfers and kernel calls are placed in the default command queue for the device
  - For CUDA, this is initially the NULL stream
  - For OpenCL, this is the command queue created with the device initialization
  - The default command queue can be changed:

```
dev.push_command_queue();
dev.set_command_queue(1);
```

Geryon

Vector and Matrix Containers

Geryon Vector/Matrix

- Containers for host, device, and texture
  - UCL_H_Vec, UCL_H_Mat, UCL_D_Vec, UCL_D_Mat, UCL_Image
- In addition to the usual conveniences, the separate type for matrices allows the library to maintain the correct byte-alignment for each row (efficient global memory access)
  - The library is lenient and allows copies from vector to matrix types and vice versa

Geryon Vector/Matrix

- Functions operating on containers make use of templates and enumerated traits
  - Almost all branches in library can be eliminated at compile time
  - Most routines require no additional overhead
- Currently all containers are row-major

Geryon Vector/Matrix

- When allocating memory, a device object is passed
  - Know where to allocate the memory
  - The container stores a smart pointer to the default command queue for the device
    - Unless a command queue is explicitly passed for copy, print, zero, etc. routines, the default command queue is used

Geryon – Host Allocation

```
// Allocate page-locked memory (2 rows, 3 columns)
UCL_H_Mat<float> h_mat;
h_mat.alloc(2,3,dev);
// --- or ---
UCL_H_Mat<float> h_mat(2,3,dev);

// If you want to explicitly specify the allocation type:
// - set kind=UCL_RW_OPTIMIZED for page-locked
// - set kind=UCL_WRITE_OPTIMIZED for write-combined
// - set kind=UCL_NOT_PINNED for standard malloc
h_mat.alloc(2,3,dev,kind);
// --- or ---
UCL_H_Mat<float> h_mat(2,3,dev,kind);

// Allocate page-locked vector with 6 elements
UCL_H_Vec<float> h_vec(6,dev);
// --- or ---
h_vec.alloc(6,dev);
// …
```

Geryon – Device Allocation

```
// Allocate device memory (2 rows, 3 columns)
UCL_D_Mat<float> d_mat;
d_mat.alloc(2,3,dev);
// --- or ---
UCL_D_Mat<float> d_mat(2,3,dev);

// You can specify that the memory will be read- or
// write-only within a device kernel
// -kind=UCL_READ_ONLY, UCL_WRITE_ONLY, or UCL_READ_WRITE
d_mat.alloc(2,3,dev,kind);
// --- or ---
UCL_D_Mat<float> d_mat(2,3,dev,kind);

// Allocate device vector with 6 elements
UCL_D_Vec<float> d_vec(6,dev);
// --- or ---
d_vec.alloc(6,dev);
// …
```

Geryon - Views

- The containers can also use existing memory allocations on the host and device instead of allocating new memory
  - When the container is "viewing" another allocation, it will not free the memory allocations

Geryon – Views

```
// Allow access to an existing Geryon memory allocation
mat.view(existing_mat);
// Allow access to first nelements of Geryon allocation
mat.view(existing_mat,n);
// Allow access to slice of Geryon allocation
mat.view(existing_mat,rows,cols);
// To use a user defined row stride (pitch) in elements
mat.view(existing_mat,rows,cols,stride);

// Allow access to an existing CUDA or OpenCL allocation
// - For double, ptr is double*, CUdeviceptr, or cl_mem
// - n is number of elements (not bytes)
// - For views of existing device memory, restricted
//   to API for that pointer (CUdeviceptr, cl_mem, etc)
// - Must pass a device
mat.view(ptr,n,dev);              // -- or –
mat.view(ptr,rows,cols,dev);      // -- or –
mat.view(ptr,rows,cols,stride,dev);
```

Geryon – Views

```
// A view may start at a specified offset (in elements
// not bytes) from the beginning of an existing
// allocation
mat.view_offset(offset,existing_mat);           // -or-
mat.view_offset(offset,existing_mat,n);         // -or-
mat.view_offset(offset,existing_mat,rows,cols); // -or-
mat.view_offset(offset,existing_mat,rows,cols,stride);
mat.view_offset(offset,ptr,n,dev);              // -or-
mat.view_offset(offset,ptr,rows,cols,dev);      // -or-
mat.view_offset(offset,ptr,rows,cols,stride,dev);

// When "viewing" Geryon containers, use the row_size()
// function to determine the offset of device matrices
mat.view_offset(mat.row_bytes()*nrows,mat);
```

Geryon – Resizing Allocations

```
// You can resize host and device allocations using
// the resize member function.
mat.resize(numel);            // -or-
mat.resize(rows,cols);

// To resize only if the specified size is bigger than
// the current allocation, use resize_ib (if bigger)
mat.resize_ib(numel);        // -or-
mat.resize_ib(rows,cols);
```

Geryon – Zero and Free

```
// You can set every element to zero on host or device
// -- This is currently not optimized for OpenCL and
//    should not be used in a loop
h_mat.zero();
d_mat.zero();

// To zero the first n elements (not bytes)
h_mat.zero(n);
d_mat.zero(n);

// Free host or device memory
// - Memory is automatically freed on destruction
// - If alloc() is called, any previous memory is freed
h_mat.clear();
d_mat.clear();
```

Geryon – Command Queues

```
// You can access the default command queue (stream)
// associated with a container
command_queue &cq=h_mat.cq();

// If you do not have access to a device object for
// memory allocation, you can use another matrix or
// vector object to allocate the memory on the same
// device with the same command queue
h_mat.alloc(2,3,h_vec);

// The command queue executes asynchronously on the
// device while host code is running
// You can block until the default command queue
// associated with container is done with all commands.
h_mat.sync();
```

Geryon – Host element access

```
// You can use 1D and 2D indexing on both vector and
// matrix types
cout << "The 4th element is: " << h_mat(3) << endl;
// --- or ---
cout << "The 4th element is: " << h_mat(1,0) << endl;

// Setting the 4th element to zero
h_mat(3)=0;
// --- or ---
h_mat(1,0)=0;

// Get the pointer to the beginning of the array
float *h_begin=h_mat.begin();
// Get the pointer to one past the last element
float *h_end=h_mat.end();
```

Geryon – Device element access

```
// Currently device element access is limited to kernels
// run on the device. Direct element accessing for
// debugging purposes might be added in future versions

// To access the API type that should be passed to a
// kernel for array access use the begin() member
add_kernel_param(&d_mat.begin());
```

Geryon – Vector/Matrix Size

```
// The following accessors work for all vector/matrix and
// host/device containers:
cout << “The matrix has “ << h_mat.numel()
     << “elements, “ << h_mat.rows() << “ rows, and “
     << h_mat.cols() << “ columns.\n”;

// Because some containers use extra columns for padding
// the row_size() and row_bytes() accessors can be used
// to determine the stride for a row in elements and
// bytes respectively.
cout << “The number of columns is: “ << h_mat.cols()
     << “ and the number of columns with padding is: “
     << h_mat.row_size() << endl;

// The pointer for row i and column j is:
h_mat.begin()+h_mat.row_size()*i+j
```

Geryon

Memory Copy Routines

Geryon – ucl_copy

- ucl_copy is the copy routine for all Geryon containers
- The function is passed an *async*argument for any copy
  - if (*async*==false) perform blocking copy
  - if (*async*==true) perform asynchronous copy using default command queue on destination container
  - if (*async*==command_queue) perform asynchronous copy using the specified command_queue
  - The argument is ignored for some copies (e.g. host->host copy cannot be asynchronous)

Geryon – ucl_copy

- If no size parameter is passed to ucl_copy, the entire source container is passed
- If 1 size parameter is passed to ucl_copy, this parameter is interpreted as a number of elements
- If 2 size parameters are passed, this parameter is interpreted as the number of rows and columns
- The routine understands and handles padding for device matrices correctly

Examples **(page 1)**

(x's represent alignment padding - to maintain alignment)
(o's represent a larger matrix in memory)
(vectors represented as single row)

```
-----------------------------------------------------------
 dst          src             command
-----------------------------------------------------------
 0 1 2 3 4 <-- 0 1 2 3 4        ucl_copy(dst,src,async)

 0 1 2 3  <-- 0 1 2 3 4         ucl_copy(dst,src,4,async)

 0 1 2    <-- 0 1 2 3 4 5       ucl_copy(dst,src,async)
 3 4 5

 0 1 2 3 4 5 <-- 0 1 2          ucl_copy(dst,src,async)
              3 4 5

 0 1 2    <--  0 1 2          ucl_copy(dst,src,async)
 3 4 5          3 4 5

 0 1 2    <--  0 1 2          ucl_copy(dst,src,6,async)
 3 4 5          3 4 5
              5 6 7

 0 1 2    <--  0  1  2  3     ucl_copy(dst,src,2,3,async)
 4 5 6          4  5  6  7
              8  9  10 11
```

Examples **(page 2)**

(x's represent alignment padding - to maintain alignment)
(o's represent a larger matrix in memory)
(vectors represented as single row)

```
-------------------------------------------------------------
dst           src            command
-------------------------------------------------------------
0 1 2 x x  <--  0 1 2          ucl_copy(dst,src,async)
3 4 5 x x       3 4 5

0 1 2      <--  0 1 2 x x      ucl_copy(dst,src,async)
3 4 5           3 4 5 x x

0 1 2 o o  <--  0 1 2          ucl_copy(dst,src,2,3,async)
3 4 5 o o       3 4 5
o o o o o

0 1 2 o o  <--  0 1 2 3 4 5    ucl_copy(dst,src,2,3,async)
3 4 5 o o
o o o o o

0 1 o o o  <--  0 1 2 3 4 5    ucl_copy(dst,src,2,2,async)
2 3 o o o
o o o o o
```

Examples **(page 3)**

(x's represent alignment padding - to maintain alignment)
(o's represent a larger matrix in memory)
(vectors represented as single row)

```
 ------------------------------------------------------------
 dst              src           command
 ------------------------------------------------------------

  0 1 2 o o    <--  0  1  2  3  4    ucl_copy(dst,src,2,3,async)
  5 6 7 o o         5  6  7  8  9
  o o o o o         10 11 12 13 14

  0 1 2 5 6 7  <--  0  1  2  3  4    ucl_copy(dst,src,2,3,async)
               5  6  7  8  9
               10 11 12 13 14
```

Geryon – ucl_copy

- If it is desired to copy using a non-zero offset from the beginning of a source or destination allocation, use *views*
- If it is desired to copy to a source or destination allocation that was not performed within Geryon, use *views*

Geryon – type casting

- Type casting is automatically performed by ucl_copy when the source and destination matrices differ
- When type casting is performed, a cast buffer is allocated
  - For casting within a loop, it is more efficient to allocate a cast buffer only once and pass this to the ucl_cast_copy routine
    - For S-Objects, this is not necessary; the cast buffer is only allocated when the s-object is allocated.

Geryon

I/O

```
// Print the entire vector matrix to standard out
cout << d_mat << endl;
// --- or ---
ucl_print(d_mat);
// Print first n elements of vector or matrix
ucl_print(d_mat,n);
// Print first n elements of vector to stderr
ucl_print(d_mat,n,cerr);
// Print first n elements of vector with comma delim
ucl_print(d_mat,n,cerr,”,”);

// Print upper 2x3 slice of matrix
ucl_print(d_mat,2,3);
// Print upper slice with , delimeter and ; at row end
ucl_print(d_mat,2,3,cout,”,”,”;\n”);

// !!! For const qualified containers, a device object
// !!! must be passed as the last argument of ucl_print
ucl_print(d_mat,2,3,dev);
```

Geryon

S-Objects

Geryon – S-Objects

- S-objects can be used to make the code cleaner/succinct when a vector/matrix on the host and the device store the same data and require frequent synchronization
  - Single object for both host and device storage
  - Different types (e.g. float/double) can be used on the host and the device and type casting is performed automatically
  - When the device is a CPU or shares memory with the host, a single allocation is used and all copies/synchronizations are ignored (type casting can still require another allocation of a separate buffer)
  - Fewer lines of code required versus separate host and device objects
  - The separate host and device objects are public members
    - The UCL_Vector S-object has a UCL_H_Vec and UCL_D_Vec object
    - The UCL_Matrix S-object has a UCL_H_Mat and UCL_D_Mat object

Geryon – S-Objects

```
// Allocate vector of length 6 with double precision on
// the host and single precision on the device
UCL_Vector<double,float> my_vec(6,dev);

// To access the host vector (e.g. to print)
std::cout << my_vec.host << std::endl;

// To access the device vector (e.g. to print)
std::cout << my_vec.device << std::endl;

// Allocate 2x3 matrix with double precision on
// the host and single precision on the device
UCL_Matrix<double,float> my_mat(2,3,dev);

// To access the host vector (e.g. to print)
std::cout << my_mat.host << std::endl;

// To access the device vector (e.g. to print)
std::cout << my_mat.device << std::endl;
```

Geryon – S-Objects

```
// To get memory usage (including any buffers)
std::cout << “Host memory usage: “
        << my_vec.host_mem_usage << “ bytes\n”;
std::cout << “Device memory usage: “
        << my_vec.dev_mem_usage << “ bytes\n”;
```

Geryon – Equivalent Code

S-Object	**Separate Host/Device Objects**

UCL_Vector<double,float> my_vec;	UCL_H_Vec<double> h_vec;
	UCL_D_Vec<float> d_vec;

my_vec.alloc(6,dev);	h_vec.alloc(6,dev);
	d_vec.alloc(6,dev);

UCL_Vector<double,float> my_vec(6,dev);	UCL_H_Vec<double> h_vec(6,dev);
	UCL_D_Vec<float> d_vec(6,dev);

UCL_Vector<double,float>
my_vec(6,dev,UCL_RW_OPTIMIZED,UCL_READ_ONLY);
	UCL_H_Vec<double>
	h_vec(6,dev,UCL_RW_OPTIMIZED);
	UCL_D_Vec<float> d_vec(6,dev,UCL_READ_ONLY);

my_vec.alloc(6,dev,UCL_RW_OPTIMIZED,UCL_READ_O
NLY);
	h_vec.alloc(6,dev, UCL_RW_OPTIMIZED);
	d_vec.alloc(6,dev,UCL_READ_ONLY);

UCL_Vector<float,float> my_vec(6,dev);	h_vec.alloc(6,dev);
	if (dev.shared_memory())
	    d_vec.view(h_vec);
	else
	    d_vec.alloc(6,dev);

UCL_Matrix<double,float> my_mat(2,3,dev);	UCL_H_Mat<double> h_mat(2,3,dev);
	UCL_D_Vec<float> d_vec(2,3,dev);

my_vec.clear();	h_vec.clear();
	d_vec.clear();

// etc…	// etc…

Geryon – Equivalent Code

S-Object	**Separate Host/Device Objects**
my_vec.update_host();	ucl_copy(h_vec,d_vec,true);
my_vec.update_device();	ucl_copy(d_vec,h_vec,true);
my_vec.update_host(async);	ucl_copy(h_vec,d_vec,async);
my_vec.update_device(async);	ucl_copy(d_vec,h_vec,async);
my_vec.update_host(6,async);	ucl_copy(h_vec,d_vec,6,async);
my_mat.update_device(2,3,async);	ucl_copy(d_vec,h_vec,2,3,async);
// etc…	// etc…

Geryon – S-Objects

- The following member functions call the equivalent function on the host allocation:
  - cq(), sync(), rows(), cols(), numel(), operator[], operator(), row_size(), row_bytes(), element_size()
- ucl_print and operator << work on the host allocation when the S-object is used
- The zero() member function zeros the memory on the host and the device.
- Pointers to S-Objects can be passed as arguments to kernels.
  - Same as passing &mat.device.begin()

Geryon

Device Timing and Advanced Synchronization

```
UCL_Timer timer;

// Initialize a timer for use on the default queue
timer.init(dev);
// --- or --- initialize for specific command queue
timer.init(dev,dev.cq());

// Time all commands added to command queue after start()
// and before stop()
timer.start();
// …
timer.stop();

// Block on the host until all commands in the command
// queue up to stop() have completed and return time
double milliseconds=timer.timer();
// --- or ---
double seconds=timer.seconds();

// For multiple timings, init() need only be called once
// For multiple timings, timer() or seconds() must be
//  called before issuing another start() command
```

```
UCL_Timer timer_a(dev), timer_b(dev);

// To synchronize by blocking to a specific point in
// the command queue, use timer objects:

timer_a.start();
ucl_copy(dest1,src1,true);
timer_a.stop();
timer_b.start();
ucl_copy(dest2,src2,true);
timer_b.stop();

// To block on cpu until the first copy has finished:
timer_a.sync_stop();
//     --- or ---
timer_b.sync_start();
```

Geryon

Vector Add Example

```
#include “nvc_device.h”
#include “nvc_mat.h”
#include “nvc_timer.h”
using namespace std;
using namespace ucl_cudart;

int main() {
  UCL_Device dev;
  if (dev.num_devices()==0) exit(1);
  dev.set(0);

  UCL_H_Vec<double> a(6,dev), b(6,dev);
  UCL_D_Vec<float> dev_a(6,dev), dev_b(6,dev);
  UCL_D_Vec<float> answer(6,dev);
  UCL_Timer timer_com(dev), timer_kernel(dev);

  for (inti=0; i<6; i++) { a[i]=i; b[i]=i; }

  timer_com.start();
  ucl_copy(dev_a,a,true);
  ucl_copy(dev_b,b,true);
  timer_com.stop();

  timer_kernel.start();
  // Call kernel here with dev_a.begin(), dev_b.begin(), answer.begin()
  timer_kernel.stop();

  cout << “Answer: “ << answer << endl << “Input copy time: “
       << timer_com.seconds() << endl << “Kernel time: “
       << timer_kernel.seconds() << endl;

  return 0;
}
```

Geryon

Vector Add Example using S-objects

```
#include “nvc_device.h”
#include “nvc_mat.h”
#include “nvc_timer.h”
using namespace std;
using namespace ucl_cudart;

int main() {
  UCL_Device dev;
  if (dev.num_devices()==0) exit(1);
  dev.set(0);

  UCL_Vector<double,float> a(6,dev), b(6,dev);
  UCL_D_Vec<float> answer(6,dev);
  UCL_Timer timer_com(dev), timer_kernel(dev);

  for (inti=0; i<6; i++) { a[i]=i; b[i]=i; }

  timer_com.start();
  a.update_device();
  b.update_device();
  timer_com.stop();

  timer_kernel.start();
  // Call kernel here with a, b, and answer
  timer_kernel.stop();

  cout << “Answer: “ << answer << endl << “Input copy time: “
       << timer_com.seconds() << endl << “Kernel time: “
       << timer_kernel.seconds() << endl;

  return 0;
}
```

Geryon

Kernel Management

Geryon – Program/Kernel

- A *program* in Geryon is 1 or more kernel functions contained in a single file or string
  - Program objects are for loading and compiling kernels
  - OpenCL can compile from source, CUDA must use PTX or binary
- A *kernel* in Geryon is a single kernel from a program
  - Kernel objects are for running kernels on a device

```
// Program takes context & default command queue from dev
UCL_Program program(dev);

// Read a program from file and compile
// - Flags are currently ignored for CUDA
// - If flags==“BINARY”, no compilation is performed
// - For CUDA Runtime, load is ignored; link with the
//   kernel object instead
string flags=“-cl-fast-relaxed-math –D Scalar=float”;
program.load(“foo.cl”,flags.c_str());
// --- or --- read program from const char * string
program.load_string(foo_str,flags.c_str());
// --- or --- load a precompiled binary (currently CUDA)
program.load_binary(filename);

// To get a build log associated with compilation
// - For CUDA Runtime, load is ignored; link with the
//   kernel object instead
string clog;
program.load(“foo.cl”,flags.c_str(),&clog);
// --- or ---
program.load_string(foo_str,flags.c_str(),&clog);
```

```
// Get a kernel object that uses the vec_add function
// in the program.
// - Takes default command queue from the program
UCL_Kernel k_vec_add(program,”vec_add”)

// Set function arguments to vec_add
k_vec_add.set_arg(0,&dev_a.begin());
k_vec_add.set_arg(1,&dev_b.begin());
k_vec_add.set_arg(2,&answer.begin());
// --- or --- add up to 30 args at a time
k_vec_add.add_args(&dev_a.begin(),&dev_b.begin(),
                   &answer.begin());

// Set up 1-dimensional execution grid to add 6 elements
size_t num_blocks=2, block_size=3;
k_vec_add.set_size(num_blocks,block_size);

// Enqueue the kernel in the default command queue
k_vec_add.run();
```

```
// To set up a 2-dimensional kernel execution grid
k_vec_add.set_size(num_blocks_x,num_blocks_y,
                   block_size_x,block_size_y);
// or to set up a 2-dimensional kernel execution grid
// with 3-D blocks
k_vec_add.set_size(num_blocks_x,num_blocks_y,
                   block_size_x,block_size_y,block_size_z);

// To run in a command queue other than the default, pass
// the command queue in set_size()
command_queue &vec_add_q=dev.cq(1);
k_vec_add.set_size(num_blocks,block_size,vec_add_q);
k_vec_add.run();
```

```
// To clear any arguments associated with a kernel
k_vec_add.clear_args();

// To clear any previous args, add arguments and run
// in a single command
// - note: requires size to be set
k_vec_add.run(&dev_a.begin(),&dev_b.begin(),
          &answer.begin());

// You can now also pass the containers directly as
// arguments to a kernel (the begin() is implicit)
k_vec_add.run(&dev_a,&dev_b,&answer);
```

Pitfalls

- There is no type casting between arguments passed with set_arg(), add_args(), run() and the arguments accepted by the kernel
    - Be carefull of size_t/int mismatch, etc.

Geryon

Writing Kernels that are OpenCL and CUDA Compatible

OpenCL Notes

- To allow double precision on supported devices, you must include in the kernel text:
  - #pragma OPENCL EXTENSION cl_khr_fp64 : enable
- Atomic functions are also optional
- Inlined kernel functions do not need to be qualified
  - static __inline__ __device__ void foo()
    - in CUDA will just be
  - inline void foo()

Preprocessor definitions

- To allow the same kernel text to compile with both CUDA and OpenCL, preprocessor directives can be used to handle differences in built in function and type names such as accessing the thread index.
- The file ucl_nv_kernel has some definitions that can help with this
- Example: define NV_KERNEL only when compiling kernel with CUDA… (next page)

```
#ifdef NV_KERNEL

#define __global
#define GLOBAL_ID_X threadIdx.x+INT_MUL(blockIdx.x,blockDim.x)
#define GLOBAL_ID_Y threadIdx.y+INT_MUL(blockIdx.y,blockDim.y)
#define THREAD_ID_X threadIdx.x
#define THREAD_ID_Y threadIdx.y
#define BLOCK_ID_X blockIdx.x
#define BLOCK_ID_Y blockIdx.y
#define BLOCK_SIZE_X blockDim.x
#define BLOCK_SIZE_Y blockDim.y
#define __kernel extern “C” __global__
#define __local __shared__
#define mul24 __mul24

#else

#define GLOBAL_ID_X get_global_id(0)
#define GLOBAL_ID_Y get_global_id(1)
#define THREAD_ID_X get_local_id(0)
#define THREAD_ID_Y get_local_id(1)
#define BLOCK_ID_X get_group_id(0)
#define BLOCK_ID_Y get_group_id(1)
#define BLOCK_SIZE_X get_local_size(0)
#define BLOCK_SIZE_Y get_local_size(1)
#define __syncthreads() barrier(CLK_LOCAL_MEM_FENCE)

#endif
```

No templates in OpenCL

- Compile kernel with flags that define template arguments "-D Scalar=float –D Ordinal=int" instead

```
#ifdef NV_KERNEL
template <class Scalar, class Ordinal>
#endif
__kernel void vec_add(DEV_PTR *a, DEV_PTR *b, DEV_PTR *ans) {
  Ordinal i=GLOBAL_ID_X;
  ans[i]=a[i]+b[i];
}
```

- The string for a given data type can obtained using ucl_template_name<>()
  - string flag=string("-D Scalar=")+ucl_template_name<Scalar>();

Geryon

Running OpenCL Kernels on CPUs

Don’t need device allocations or host-device copies

- Use views:

```
UCL_H_Vec host_mem;
host_mem.alloc(6,dev);
UCL_D_Vec device_mem;
// Only allocate device memory if the device is not a CPU
if (dev.device_type==UCL_CPU)
  device_mem.view(host_mem);
else
  device_mem.alloc(6,dev);
```

- ucl_copy will do nothing if the pointer/offset for the source and destination are the same (if a blocking copy is specified, the routine will still block until the command queue is sync’d
- S-objects handle this automatically (no device allocations if possible)
- For the AMD OpenCL library, set the CPU_MAX_COMPUTE_UNITS environment variable to control the number of cores used for kernel execution

Pitfalls

- When viewing host allocations from device containers
  - Make sure the floating precision is the same for both allocations (automatic type casting will no longer be used)
  - Be careful about asynchronous copies in the code
    - Hiding host-device communications often assumes 2 buffers (regardless of whether or not they are in the same address space)

Geryon

Error Handling

Geryon – CUDA and OpenCL Errors

- No error checking for out-of-bounds indexing
- Most CUDA and OpenCL errors are handled with output to stderr and exit.
  - Most of these should be handled by the software developer outside of Geryon
- The current exceptions are device initialization, kernel compilation and memory allocation
  - For these functions, error flags and/or build logs are returned that can be handled by the developer
  - Preprocessor flags must be set if the developer does not want Geryon to handle the errors

Geryon – Preprocessor Directives

- Define
  - UCL_DEBUG – turn on some sanity and mat/vec size checks (mostly assert statements)
    - Also defines UCL_SYNC_DEBUG and UCL_DESTRUCT_CHECK
  - UCL_NO_EXIT – do not exit with a message to stderr when an error occurs in device initialization, kernel compilation, or memory allocation
  - UCL_NO_API_CHECK – turn off error checking for CUDA and OpenCL API calls that do not involve memory allocation or kernel compilation *(unlikely to improve performance)*
  - UCL_SYNC_DEBUG – Block after each device call and check for error (*currently blocking will only be performed for CUDA*)
  - UCL_DESTRUCT_CHECK – Error checks are not performed by default for calls that free memory, contexts, events, etc. This will enable error checking for these calls.

```
// To check for device initialization errors
// -- Define UCL_NO_EXIT preprocessor directive
if (dev.set(0)!=UCL_SUCCESS)
  cerr << “Could not initialize device for use by program.”



// To check for memory allocation errors
// -- Define UCL_NO_EXIT preprocessor directive
if (d_mat.alloc(6,dev)!=UCL_SUCCESS) {
  cerr << “Could not allocate 6 elements on device “
       << dev.name() << endl;
}
```

```
// To check for compiling errors
// -- Define UCL_NO_EXIT preprocessor directive
// -- Note: similar output is given by Geryon if
//    UCL_NO_EXIT is not defined
string clog;
int err=program.load(“test.ocl”,””,&clog);
if (err!=UCL_SUCCESS) {
  if (err==UCL_FILE_NOT_FOUND)
    cerr << “Could not find test.ocl\n”;
  if (err==UCL_COMPILE_ERROR)
    cerr << “Problem compiling test.ocl:\n”
         << clog << endl;
  exit(1);
}
if (kernel.set_function(program,”foo”)!=UCL_SUCCESS)
  cerr << “Could not find function foo in test.ocl\n”;
```

Geryon

1D Texture Binding

1D Texture Binding

- In CUDA, can bind existing memory allocations to textures for constant access that can be cached
  - In Geryon, this is accomplished by the UCL_Texture class
  - This feature is not currently in OpenCL, so UCL_Texture routines in the opencl namespace do nothing.
  - For textures, a vector length is specified; each fetch grabs a vector of the specified length

Assigning a texture reference

- CUDA Driver/OpenCL

```
#include “nvd_texture.h”

// Texture must be defined within a Program
UCL_Texture my_tex(my_program,”texture_name”);
// -- or –
UCL_Texture my_tex;
my_tex.get_texture(my_program,”texture_name”);
```

- CUDA Runtime

```
#include “nvc_texture.h”
// Texture must be defined within same file
texture<float2> tex;

UCL_Texture my_tex(tex);
// -- or –
UCL_Texture my_tex;
my_tex.get_texture(tex);
```

Binding a texture

```
// To bind a float array with vector length 2
my_tex.bind_float(my_array.begin(),2);

// Texture is fetched within a kernel using
// tex1Dfetch(tex,i);

// To unbind the array
my_tex.unbind();
```

Geryon

Current Status and Future Work

Current Status

- Texture/Image Containers are not implemented
- Library tests are in place for common use, but not all routines/combinations are covered
- zero() routine for OpenCL is inefficient
- Efficiency of 2D copy in OpenCL is lagging
- Need to test most efficient type cast procedure (host or device cast)
  - Currently type casting is performed on the host
  - For archs that support double precision, other options
- Need column-major/transpose copies