Brown Deer Technology

The COPRTHR Primer

revision 1.2

The CO-PRocessing THReads (COPRTHR) SDK provides OpenCL™ related libraries and tools for developers targeting GPU compute technology and hybrid CPU/GPU computing architectures.

Contents

---

1 Release Notes

1.1 New In Version 1.2 (middletown release)

- Expanded Operating System support:
    - Full SDK support for FreeBSD-8 including an open-source OpenCL implementation (libocl) for amd64
    - STDCL beta support for Windows 7 / MSVS 2010
- C++ container classes with OpenCL device-sharable memory:
    - OpenCL extension of STL vector and BOOST multi_array
    - Containers allow conventional data management on the host side with memory synchronization for OpenCL devices
- Transparent/automatic GPU acceleration of vector operations for C++ containers
    - Automatic kernel generation and host interfacing uses CLETE expression template engine
- Improvements to the SDTCL interface including
    - Support for image2D memory allocation using clmalloc
    - CLGL buffer sharing support
    - Run-time device management including exclusive device locks for MPI support
- Improvements to the open-source OpenCL run-time (libocl) including
    - Partial support for images
    - Many enhancements for performance and functionality
- Updated examples and documentation
- STDCL now tested against AMD SDK v2.4, Nvidia CUDA 4.0, Intel OpenCL SDK v1.1

1.2 Support and Requirements

With this release, support has been expanded to included FreeBSD-8 and Windows 7 operating systems, with Windows support limited to the basic functionality provided by the STDCL interface to OpenCL. Support continues for most modern Linux distributions including RHEL 5.4/5.5, CentOS 5.4/5.5, OpenSuSE 11.2/11.3 and Ubuntu 10.4. Specific feature support by operating system is shown in the table below.

COPRTHR Feature	Description	Linux	FreeBSD	Windows
libstdcl	STDCL interface	x	x	x
cltrace	tracing tool	x	x	
clld	link tool	x	x	
libocl	OpenCL x86_64 runtime	x	x	

This release is compatible with the OpenCL implementations provided with AMD APP v2.4, Nvidia CUDA-4 and Intel OpenCL SDK v1.1. In addition, an open-source OpenCL run-time implementation for x86_64 multi-core processors is provided as part of the COPRTHR SDK, which may be used on platforms for which no vendor support is available. The COPRTHR OpenCL implementation may also be of interest since it exhibits better performance than vendor implementations on some real-world benchmarks.

This release supports x86_64 CPUs from AMD and Intel as well as GPUs from AMD and Nvidia, and has been tested successfully on the following graphics cards: AMD Radeon HD 4850, 4870, 4870X2, 5870, 5970, 6970, AMD FirePro V8800, Nvidia Tesla S1070, C2050, and C2070.

If you are only interested in the basic functionality of the STDCL interface for OpenCL, no additional packages are required beyond the standard vendor implementation of OpenCL for your platform. (If none is available, try the open-source implementation provided by with COPRTHR SDK.)

If you wish to use the `clld` tool for embedding OpenCL kernel code into ELF objects to create single executables, you will need to install libelf-0.8.13. Note that versions of libelf typically found on Linux distributions (designated 1.x) are *not compatible* and not very useful since they often contain many undocumented behaviors, and should not be used.

If you wish to use the open-source OpenCL run-time implementation provided by COPRTHR SDK you will need to install libelf-0.8.13 along with a few additional packages. Specifically, you will need LLVM and CLANG v2.6 and the ATI Stream SDK v2.1. (Newer versions of the AMD SDK are *not* valid substitutes.)

Note that for FreeBSD, the versions of LLVM and libelf provided in the ports collection are not compatible and a direct installation of these packages is required. In addition, the clc executable provided by the ATI Stream SDK v2.1 must be properly branded as a Linux executable and run under Linux binary compatibility. See the FreeBSD Handbook for more details (read the section on Linux binary compatibiliy).

The table below provides a comprehensive matrix of required packages matched to a specific platform and feature set.

COPRTHR Feature				Package	Download
libstdcl	cltrace	clld	libocl		
Linux Red Hat 5.4/5.5, CentOS 5.4/5.5, OpenSuSE 11.2					
O				AMD APP v2.4	http://developer.amd.com/sdks/AMDAPPSDK/downloads
O				Nvidia CUDA 4	http://developer.nvidia.com/cuda-toolkit-40

O				Intel OpenCL SDK-1.1	http://software.intel.com/en-us/articles/download-intel-opencl-sdk
		R	R	libelf 0.8.13	http://www.mr511.de/software/libelf-0.8.13.tar.gz
			R	LLVM-2.6	http://llvm.org/releases/2.6/llvm-2.6.tar.gz
			R	CLANG-2.6	http://llvm.org/releases/2.6/clang-2.6.tar.gz
			R	ATI Stream SDK-v2.1	http://developer.amd.com/Downloads/ati-stream-sdk-v2.1-lnx64.tgz
FreeBSD-8					
		R	R	libelf 0.8.13	http://www.mr511.de/software/libelf-0.8.13.tar.gz
			R	LLVM-2.6	http://llvm.org/releases/2.6/llvm-2.6.tar.gz
			R	CLANG-2.6	http://llvm.org/releases/2.6/clang-2.6.tar.gz
			R	ATI Stream SDK-v2.1	http://developer.amd.com/Downloads/ati-stream-sdk-v2.1-lnx32.tgz
Windows 7					
O				AMD APP v2.4	http://developer.amd.com/sdks/AMDAPPSDK/downloads
O				Nvidia CUDA 4	http://developer.nvidia.com/cuda-toolkit-40
O				Intel OpenCL SDK-1.1	http://software.intel.com/en-us/articles/download-intel-opencl-sdk

O=Optional, R=Required

1.3 Important Notes

- The libraries `libstdcl` and `libocl` are provided with debug versions `libstdcl_d` and `libocl_d`, respectively. Linking against these libraries can be very useful for debugging as well as understanding how each library operates.
- If you install a binary release, it may not have been compiled with your preferred OpenCL implementation as a default. The most reliable way to ensure the correct implementation is used is to set the appropriate environment variable. The following are examples:

  export STDGPU_PLATFORM_NAME=nvidia
  Use the Nvidia OpenCL implementation for the stdgpu context

  export STDCPU_PLATFORM_NAME=intel
  Use the Intel OpenCL implementation for the stdcpu context

- The flag CL_EVENT_RELEASE has been removed and the flag CL_EVENT_NORELEASE has been added; the default behavior of event management calls has been changed so as to always release events unless the latter flag is used.
- The use of environment variable to control certain aspects of each default context has been changed. As an example, the environment variable STDGPU is now only checked for a 0 or 1 to determine whether the context is enabled. For more information see the revised STDCL reference manual (revision 1.2).
- The type of STDCL context pointers has been changed from CONTEXT* to CLCONTEXT* to avoid namespace collisions on Windows 7. The use of CONTEXT* is still acceptable but should be considered deprecated since its use will eventually be removed.

1.4 Frequently Asked Questions

Below are answers to frequently asked questions regarding COPRTHR SDK and STDCL.

Does STDCL require the BDT OpenCL run-time?
No. The basic installation of libstdcl.so will work with any compliant OpenCL installation including the latest implementations from AMD, Nvidia and Intel.

Will using STDCL reduce performance or limit access to OpenCL functionality?
No. STDCL is implemented as a very light-weight interface and does not restrict access to direct OpenCL and fully supports asynchronous operations across multiple devices.

Are STDCL calls simply wrappers for OpenCL calls?
No. There is a bit more to the interface than wrapping OpenCL calls. For the curious, take a look at the source code.

---

2 Installation

2.1 Installation Overview

The COPRTHR SDK may be installed from pre-compiled binaries for selected platforms or built from source. It may be necessary to install one or more 3rd-party packages as part of the installation process as described in the previous section 1.2 Support and Requirements.

For Linux and FreeBSD installation a configure script is provided to customize the package. A detailed description of the various options is described at the end of this section.

2.2 Download the Release

COPRTHR SDK is available from the github project page located at http://www.github.com/browndeer/coprthr .

2.3 Linux and FreeBSD Installation

1) From the root coprthr directory, type,

```
./configure [options]
```

2) Build the package, type,

```
gmake
```

3) Install the package, type,

```
gmake install
```

4) (Optional) In order to use the open-source OpenCL run-time the tool llvm-ex must be installed. Follow the instructions in the directory /tools/llvm-ex/

5) Set the appropriate paths to use the headers and libraries when building your own application.

2.4 Windows 7 Installation

Run the Windows installer (libstdcl-1.2-RC1_win7_install.msi) and set the appropriate paths to use the headers and library from your application.

2.5 Configure Options

When building from source the configure script supports the following options:

`--prefix=/path/to/target-install-dir`
: set the root directory for installation

`--with-amd-sdk=/path/to/amd-sdk`
: set the root directory for the AMD APP SDK. The default location searched is /usr/local/atistream .

`--with-ati-sdk-21=/path/to/ati-sdk-21`
: set the root directory for the ATI Stream SDK v2.1. The default location searched is /usr/local/atistream/ati-stream-sdk-v2.1-lnx64 .

`--with-nvidia-sdk=/path/to/nvidia-sdk`
: set the root directory for the Nvidia SDK

`--with-intel-sdk=/path/to/intel-sdk`
: set the root directory for the Intel SDK

`--enable-debug-libs`
: enable building debug version of libstdcl and libocl (enabled by default)

`--enable-cltrace`
: enable building cltrace tool (enabled by default)

`--enable-clld`
: enable building clld tool (enabled by default)

`--enable-clete`
: enable CLETE support (enabled by default)

`--enable-libocl`
: enable building experimental libocl (disabled by default)

`--enable-libocl-ngpu`
: enable multiple CPU device support in libocl (disabled by default)

`--enable-deprecated-flags`
: enable deprecated flags for backward compatibility (disabled by default)

---

3 Test Scripts

A set of test scripts are provided as a quick check to verify the installation (Linux and FreeBSD only). Passing these tests does not guarantee trouble free operation. However, failing to pass these tests provides an immediate indication that something is wrong. There are two sets of test scripts designed to test libstdcl and libocl. The tests themselves consist of kernels and C code automatically generated by a set of PERL scripts. The full suite of tests will execute close to 3,000 unique OpenCL kernels executions. There are two variants - the "test" and the "quicktest" - and it is highly advisable to use the quicktest unless you plan to let your machine run for an hour or so. After installation, the tests can be executed from the root COPRTHR directory by typing either:

```
make quicktest
```

or

```
make test
```

At this point you will see a lot of activity generating and compiling the tests. Once the tests begin to run, success or failure is easy to discern since each test

will output either [pass] or [fail] to the screen. If all tests pass, things are looking good. If you find that a test has failed, something is wrong with the installation. Inspection of the problematic test can provide useful information as to what may have went wrong. A useful trick is to recompile the test using the debug version of the library and running the test directly without the standard redirection of output to /dev/null .

Windows 7 installation can be tested using the examples provided in msvs2010/examples/ .

---

4 Hello STDCL

As with most programming, its best to begin with a hello world example that captures the most important aspects of the API or interface. This section will describe a hello stdcl program that provides everything a programmer needs to know to get started with the interface. A basic understanding of OpenCL is helpful, but may not be necessary. The example will perform an (unoptimized) matrix-vector product on a GPU.

First we need kernel code to define the algorithm that will run on the GPU. This code will be stored in the file matvecmult.cl and compiled at run-time using the just-in-time (JIT) compilation model of OpenCL. The kernel is executed for each thread in the workgroup spanning the execution range which consists of each element in the output vector.

```
/* matvecmult.cl */

__kernel void matvecmult_kern(
   uint n,
   __global float* aa,
   __global float* b,
   __global float* c
)
{
   int i = get_global_id(0);
   int j;
   float tmp = 0.0f;
   for(j=0;j<n;j++) tmp += aa[i*n+j] * b[j];
   c[i] = tmp;
}
```

Next, we need host-code to run on the CPU and manage the execution on the GPU. The host code below contains everything needed to executute the above kernel. By using STDCL this program is many times smaller than what would be required to use OpenCL directly, and its also a lot simpler based on the use of better syntax and semantics.

```
/* hello_stdcl.c */

#include <stdio.h>
#include <stdcl.h>

int main()
{
   cl_uint n = 64;

   /* use default contexts, if no GPU use CPU */
   CLCONTEXT* cp = (stdgpu)? stdgpu : stdcpu;

   unsigned int devnum = 0;

   void* clh = clopen(cp,"matvecmult.cl",CLLD_NOW);
   cl_kernel krn = clsym(cp,clh,"matvecmult_kern",0);

   /* allocate OpenCL device-sharable memory */
   cl_float* aa = (float*)clmalloc(cp,n*n*sizeof(cl_float),0);
   cl_float* b = (float*)clmalloc(cp,n*sizeof(cl_float),0);
   cl_float* c = (float*)clmalloc(cp,n*sizeof(cl_float),0);

   /* initialize vectors a[] and b[], zero c[] */
   int i,j;
   for(i=0;i<n;i++) for(j=0;j<n;j++) aa[i*n+j] = 1.1f*i*j;
   for(i=0;i<n;i++) b[i] = 2.2f*i;
   for(i=0;i<n;i++) c[i] = 0.0f;

   /* define the computational domain and workgroup size */
   clndrange_t ndr = clndrange_init1d( 0, n, 64);

   /* non-blocking sync vectors a and b to device memory (copy to GPU) */
   clmsync(cp,devnum,aa,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,devnum,b,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

   /* set the kernel arguments */
   clarg_set(cp,krn,0,n);
   clarg_set_global(cp,krn,1,aa);
   clarg_set_global(cp,krn,2,b);
```

```
   clarg_set_global(cp,krn,3,c);

   /* non-blocking fork of the OpenCL kernel to execute on the GPU */
   clfork(cp,devnum,krn,&ndr,CL_EVENT_NOWAIT);

   /* non-blocking sync vector c to host memory (copy back to host) */
   clmsync(cp,0,c,CL_MEM_HOST|CL_EVENT_NOWAIT);

   /* force execution of operations in command queue (non-blocking call) */
   clflush(cp,devnum,0);

   /* block on completion of operations in command queue */
   clwait(cp,devnum,CL_ALL_EVENT);

   for(i=0;i<n;i++) printf("%d %f %f\n",i,b[i],c[i]);

   clfree(aa);
   clfree(b);
   clfree(c);

   clclose(cp,clh);
}
```

Walking through the code with brevity, the steps are as follows:

- Select a context, using GPU if available, otherwise use CPU as a fallback.
- Open the file containing the kernel code and build/compile it. Get a handle to the kernel we want to execute.
- Allocate device-sharable memory (using semantics for allocating memory established decades ago - no membuffers to worry about). Initialize/zero our input/output data.
- Create our "N-dimension range" over which the kernel will be executed - here N is 1024 and we use a workgroup size of 64.
- Transfer the input matrix and vector to the GPU using the semantics of a memory sync to the device (non-blocking).
- Set the kernel arguments - this is a bit ugly, but no more so than direct OpenCL.
- Next, fork the kernel execution on the GPU (non-blocking) and bring the results back to the host using another memory sync, but this time we "sync to host" (both calls non-blocking).
- At this point, depending on the underlying OpenCL implementation, nothing may have actually happened, but calling clflush() will get everything on the device going.
- At this point we (cl)wait for all of our non-blocking calls to finish.
- All that is left is to print the results and cleanup the allocations we created.

---

5 More STDCL Examples

This section contains more examples designed to explain special behaviors or special features as opposed to typical use cases for "ordinary" STDCL code. The Hello STDCL code example contains almost everything the average programmer needs to use the STDCL interface.

Please note that most of these examples will *not* work with Windows 7, in some case due to lack of support for the relevant feature, and in some cases due to deficiencies with Microsoft's interpretation of the C++ standard. However, the features in most of these examples are advanced and go beyond the basic STDCL interface which will work with Windows 7.

5.1 clopen_example

STDCL provides different ways to manage OpenCL kernel code. The examples in clopen_example/ demonstrate the functionality. First we need some kernel code to use, and so we will start with a simple outer product kernel with macro defining a coefficient to included in the operation. In the kernel code below, note that if COEF is note defined it will be set to 1.

```
/* outerprod.cl */

#ifndef COEF
#define COEF 1
#endif

__kernel void outerprod_kern(
   __global float* a,
   __global float* b,
   __global float* c
)
{
   int i = get_global_id(0);
```

```
   c[i] = COEF * a[i] * b[i];
}
```

In the first example of the use of clopen() the simplest use case is shown. The file outerprod.cl is assumed to be available in the run directory for just-in-time (JIT) compilation. In the host code below, clopen() is given the filename containing the kernel code, which it will open and compile, returning a handle to the result in a manner patterned after the Linux dynamic loader call dlopen(). The kernel program is compiled and built immediately, and a subsequent call to clsym() returns the actual kernel object o opaque type cl_kernel, ready to use in subsequent OpenCL calls.

```
/* clopen_example1.c */

#include <stdio.h>
#include <stdcl.h>

int main()
{
   cl_uint n = 1024;

   /* use default contexts, if no GPU use CPU */
   CLCONTEXT* cp = (stdgpu)? stdgpu : stdcpu;

   unsigned int devnum = 0;

   /*****************************************************************
    *** this example requires the .cl file to be available at run-time
    *****************************************************************/

   void* clh = clopen(cp,"outerprod.cl",CLLD_NOW);
   cl_kernel krn = clsym(cp,clh,"outerprod_kern",0);

   if (!krn) { fprintf(stderr,"error: no OpenCL kernel\n"); exit(-1); }

   /* allocate OpenCL device-sharable memory */
   cl_float* a = (float*)clmalloc(cp,n*sizeof(cl_float),0);
   cl_float* b = (float*)clmalloc(cp,n*sizeof(cl_float),0);
   cl_float* c = (float*)clmalloc(cp,n*sizeof(cl_float),0);

   /* initialize vectors a[] and b[], zero c[] */
   int i;
   for(i=0;i<n;i++) a[i] = 1.1f*i;
   for(i=0;i<n;i++) b[i] = 2.2f*i;
   for(i=0;i<n;i++) c[i] = 0.0f;

   /* non-blocking sync vectors a and b to device memory (copy to GPU)*/
   clmsync(cp,devnum,a,CL_MEM_DEVICE|CL_EVENT_WAIT);
   clmsync(cp,devnum,b,CL_MEM_DEVICE|CL_EVENT_WAIT);

   /* define the computational domain and workgroup size */
   clndrange_t ndr = clndrange_init1d( 0, n, 64);

   /* set the kernel arguments */
   clarg_set_global(cp,krn,0,a);
   clarg_set_global(cp,krn,1,b);
   clarg_set_global(cp,krn,2,c);

   /* non-blocking fork of the OpenCL kernel to execute on the GPU */
   clfork(cp,devnum,krn,&ndr,CL_EVENT_NOWAIT);

   /* block on completion of operations in command queue */
   clwait(cp,devnum,CL_ALL_EVENT);

   /* non-blocking sync vector c to host memory (copy back to host) */
   clmsync(cp,0,c,CL_MEM_HOST|CL_EVENT_NOWAIT);

   /* block on completion of operations in command queue */
   clwait(cp,devnum,CL_ALL_EVENT);

   for(i=0;i<n;i++) printf("%d %f %f %f\n",i,a[i],b[i],c[i]);

   clfree(a);
   clfree(b);
   clfree(c);

   clclose(cp,clh);
}
```

In example 2, we make use of the macro COEF to modify the outer product calculation so as to multiple the result by a fixed constant value. In order to do this we replace the previous clopen() and clsym() calls with the code shown below. Notice the flag CLLD_NOBUILD. This flag tells clopen() to defer the

compilation and build. Then we call clbuild() which allows us to pass in arbitrary compiler options that will be used in the compilation of the kernel code. In this example we define the macro COEF to the value 2. The effect is that our code will now calculate the outer product and multiply it elementwise by 2.

```
...
   /*****************************************************************
    *** this example requires the .cl file to be available at run-time
    *** and shows how to pass compiler options to the OCL compiler
    *****************************************************************/

   void* clh = clopen(cp,"outerprod.cl",CLLD_NOBUILD);
   clbuild(cp,clh,"-D COEF=2", 0);
   cl_kernel krn = clsym(cp,clh,"outerprod_kern",0);
 ...
```

The final example only works on Linux and FreeBSD. (Apologies are extended to Windows developers.) In this example we show how to create single executables which eliminate the requirement to drag around .cl files with your application. This example requires the use of the tool clld to embed OpenCL kernel code directly into an ELF object that may be linked into the final executable. The code below again replaces the clopen()/clsym() calls. Notice that instead of passing a filename to clopen(), that argument is now set to 0 (NULL). When this is done, clopen() will return a handle to the OpenCL kernel code embeded within the executable, which will be compiled and built. The clsym() is then used as before to get the desired kernel.

```
...
   /*****************************************************************
    *** This example requires .cl file to be linked into the executable
    *** using clld.  Note that clopen is called without a filename.
    *****************************************************************/

   void* clh = clopen(cp,0,CLLD_NOW);
   cl_kernel krn = clsym(cp,clh,"outerprod_three_kern",0);
 ...
```

In order to differentiate this example from previous examples, we will embed the specialized kernel code shown below which merely calculates the outer product multiplied by a factor of 3.

```
/* outerprod_three.cl */

__kernel void outerprod_three_kern(
   __global float* a,
   __global float* b,
   __global float* c
)
{
   int i = get_global_id(0);
   c[i] = 3.0f * a[i] * b[i];
}
```

So how is this OpenCL kernel code embedded into the executable? Examine the Makefile. The key step is

```
    clld --cl-source outerprod_three.cl
```

which generates the file out_clld.o that contains the OpenCL kernel source embedded as an ELF object. Then this object file is linked in to the executable just like any other object file. Its possible to see that the executable has embedded OpenCL kernel code by using the command readelf to examine the added ELF sections,

readelf -S clopen_example3.x

If you compare the output to a typical executable you will find 5 sections not normally found in executables, namely,

```
    .clprgs, .cltexts, .clprgb, .cltextb, and .clstrtab
```

These sections are used to embed the OpenCL kernel code so that clopen() can find and build the programs.

5.2 image2d_example

The following example demonstrates a non-trivial situation where one wishes to use texture memory to create fast lookup tables used by an OpenCL kernel. This is supported with STDCL using clmalloc() and clmctl(). The latter call can be used to manipulate a memory allocation created by clmalloc() and is patterned after the UNIX ioctl() call insofar as it is intended to be a generic utility to avoid the proliferation of specialized calls within the STDCL interface. The use of texture memory from within OpenCL remains somewhat clumsy from an HPC perspective, but the performance benefits it very attractive. The method for using texture memory with STDCL retains some of the awkward semantics of OpenCL, but introduces nothing further.

The kernel code below shows the use of a simple table to create a specialized matrix-vector multiply operation. The calculation is a normal matrix-vector multiple, however, in the summation a coefficient is introduced that depends on the indices i and j which are used to lookup a coefficient in a 24 x 24 table stored as a read only image2d_t type memory.

```
/* matvecmult_special.cl */

__kernel void matvecmult_special_kern(
   uint n,
```

```
    __global float* aa,
    __global float* b,
    __global float* c,
    __read_only image2d_t table
)
{
    const sampler_t sampler0
       = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_NONE | CLK_FILTER_NEAREST;

    int i = get_global_id(0);
    int j;
    float tmp = 0.0f;
    for(j=0;j<n;j++) {
       int ri = i%24;
       int rj = j%24;
       float4 coef = read_imagef(table, sampler0, (int2)(ri,rj));
       tmp += coef.x * aa[i*n+j] * b[j];
    }
    c[i] = tmp;
}
```

The host code below shows how one can allocate memory of OpenCL image2d_t type using the standard clmalloc() call along with performing modifications to the allocation using a call to clmctl(). The critical code is highlighted in red. The memory is allocated using a standard clmalloc() call with the flag CL_MEM_DETACHED. Using this flag is key since it prevents the memory from being attached to the CL context, allowing us to manipulate the allocation a bit first. We then call clmctl() with the operation CL_MCTL_IMAGE2D causing clmctl to perform an operation on the allocation so as to "mark" it as memory of type image2d_t. The arguments after the operation specification set the shape of the memory allocation, in this case its a 24 x 24 table. The final step is to attach the memory to our CL context, stdgpu in this case. The table can then be used as a kernel argument like any other global memory allocation, and the kernel can access the memory as read only image_t type.

```
/* image2d_example.c */

#include <stdio.h>
#include <stdcl.h>

int main()
{
    cl_uint n = 1024;

    /* use default contexts, if no GPU use CPU */
    CLCONTEXT* cp = (stdgpu)? stdgpu : stdcpu;

    unsigned int devnum = 0;

    void* clh = clopen(cp,"matvecmult_special.cl",CLLD_NOW);
    cl_kernel krn = clsym(cp,clh,"matvecmult_special_kern",0);

    /* allocate OpenCL device-sharable memory */
    cl_float* aa = (float*)clmalloc(cp,n*n*sizeof(cl_float),0);
    cl_float* b = (float*)clmalloc(cp,n*sizeof(cl_float),0);
    cl_float* c = (float*)clmalloc(cp,n*sizeof(cl_float),0);

    /* initialize vectors a[] and b[], zero c[] */
    int i,j;
    for(i=0;i<n;i++) for(j=0;j<n;j++) aa[i*n+j] = 1.1f*i*j;
    for(i=0;i<n;i++) b[i] = 2.2f*i;
    for(i=0;i<n;i++) c[i] = 0.0f;


    /***
     *** Create a image2d allocation to be used as a read-only table.
     *** The table will consist of a 24x24 array of float coefficients.
     *** The clmctl() call is used to set the type and shape of the table.
     *** Note that we will only use the first component of the float4 elements.
     ***/
    cl_float4* table
       = (cl_float4*)clmalloc(cp,24*24*sizeof(cl_float4),CL_MEM_DETACHED);
    clmctl(table,CL_MCTL_SET_IMAGE2D,24,24,0);
    clmattach(cp,table);

    /* initialize the table to some contrived values */
    for(i=0;i<24;i++) for(j=0;j<24;j++) table[i*24+j].x = 0.125f*(i-j);


    /* define the computational domain and workgroup size */
    clndrange_t ndr = clndrange_init1d( 0, n, 64);

    /* non-blocking sync vectors a and b to device memory (copy to GPU)*/
```

```
   clmsync(cp,devnum,aa,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,devnum,b,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,devnum,table,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

   /* set the kernel arguments */
   clarg_set(cp,krn,0,n);
   clarg_set_global(cp,krn,1,aa);
   clarg_set_global(cp,krn,2,b);
   clarg_set_global(cp,krn,3,c);
   clarg_set_global(cp,krn,4,table);

   /* non-blocking fork of the OpenCL kernel to execute on the GPU */
   clfork(cp,devnum,krn,&ndr,CL_EVENT_NOWAIT);

   /* non-blocking sync vector c to host memory (copy back to host) */
   clmsync(cp,0,c,CL_MEM_HOST|CL_EVENT_NOWAIT);

   /* block on completion of operations in command queue */
   clwait(cp,devnum,CL_ALL_EVENT);

   for(i=0;i<n;i++) printf("%d %f %f\n",i,b[i],c[i]);

   clfree(aa);
   clfree(b);
   clfree(c);

   clclose(cp,clh);
}
```

5.3 mpi_lock_example - transparent multi-GPU device management

The STDCL interface now provides run-time inter-process device management, whereby environment variables can be used to create platform behaviors for typical multi-GPU (or multi-device in general) use cases. A typical example is assigning one GPU to each MPI process on a multi-GPU platform. It is certainly possible to have the MPI processes work out for themselves who should be using a particular device on a node with multiple devices, such a solution is inelegant. STDCL provides a better way. Assume we have a platform with 2 GPUs per node and we intend to launch 2 MPI processes per node. We would like each MPI process to have its own GPU. To achieve this simply set the environment variables,

```
   export STDGPU_MAX_NDEV=1;

   export STDGPU_LOCK=31415;
```

and ensure that these environment variables are exported by mpirun. (Use the -x option.) Note that there is nothing special about the number "31415" - the lock ID can be whatever you like. The effect of these environment variables is that when the default context stdgpu is created for each process it will only contain one GPU even though two are available on the node. Further, the common lock value used by each MPI process ensures that each processes will be provided a unique GPU, i.e., the processes will not share the same device. This is despite the fact that each MPI process "thinks" that it is using devnum 0 and makes no effort in its code to try to discern which device on the platform it should use.

The example code uses the same outerprod.cl kernel code used in the clopen_example, which will not be repeated here. The host code is shown below, wherein MPI code has been added so as to allow the outer product of two vectors two be distributed across multiple MPI processes, each performing the calculation on a GPU provided to it exclusively. Notice that no where in the code is there an effort to determine which GPU should be used on a multi-GPU platform. For every processes, devnum=0.

```
/* mpi_lock_example.c */

#include <stdio.h>
#include <stdcl.h>
#include <mpi.h>

int main( int argc, char** argv )
{

   int procid, nproc;

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &procid );
   MPI_Comm_size( MPI_COMM_WORLD, &nproc );

   cl_uint n = 64;


   /* use default contexts, if no GPU fail, need one GPU per MPI proc */
   CLCONTEXT* cp = (stdgpu)? stdgpu : 0;

   if (!cp) { fprintf(stderr,"error: no CL context\n"); exit(-1); }
```

```
   unsigned int devnum = 0; /* every MPI proc thinks its using devnum=0 */

   void* clh = clopen(cp,"outerprod.cl",CLLD_NOW);
   cl_kernel krn = clsym(cp,clh,"outerprod_kern",0);

   if (!krn) { fprintf(stderr,"error: no OpenCL kernel\n"); exit(-1); }

   /* allocate OpenCL device-sharable memory */
   cl_float* a = (float*)clmalloc(cp,n*sizeof(cl_float),0);
   cl_float* b = (float*)clmalloc(cp,n*sizeof(cl_float),0);
   cl_float* c = (float*)clmalloc(cp,n*sizeof(cl_float),0);

   /* initialize vectors a[] and b[], zero c[] */
   int i;
   for(i=0;i<n;i++) a[i] = 1.1f*(i+procid*n);
   for(i=0;i<n;i++) b[i] = 2.2f*(i+procid*n);
   for(i=0;i<n;i++) c[i] = 0.0f;

   /* non-blocking sync vectors a and b to device memory (copy to GPU)*/
   clmsync(cp,devnum,a,CL_MEM_DEVICE|CL_EVENT_WAIT);
   clmsync(cp,devnum,b,CL_MEM_DEVICE|CL_EVENT_WAIT);

   /* define the computational domain and workgroup size */
   clndrange_t ndr = clndrange_init1d( 0, n, 64);

   /* set the kernel arguments */
   clarg_set_global(cp,krn,0,a);
   clarg_set_global(cp,krn,1,b);
   clarg_set_global(cp,krn,2,c);

   /* non-blocking fork of the OpenCL kernel to execute on the GPU */
   clfork(cp,devnum,krn,&ndr,CL_EVENT_NOWAIT);

   /* block on completion of operations in command queue */
   clwait(cp,devnum,CL_ALL_EVENT);

   /* non-blocking sync vector c to host memory (copy back to host) */
   clmsync(cp,0,c,CL_MEM_HOST|CL_EVENT_NOWAIT);

   /* block on completion of operations in command queue */
   clwait(cp,devnum,CL_ALL_EVENT);


   /* now exchange results */
   float* d = (float*)malloc(nproc*n*sizeof(float));

   MPI_Allgather(c, n, MPI_FLOAT, d, n, MPI_FLOAT, MPI_COMM_WORLD);

   if (procid==0)
      for(i=0;i<nproc*n;i++) printf("%d %f\n",i,d[i]);

   free(d);

   clfree(a);
   clfree(b);
   clfree(c);

   clclose(cp,clh);

   MPI_Finalize();

}
```

The trick to allowing this simplicity in the host code is to simply set the correct environment variables when the job is run. As an example, the run script run_two_gpus.sh is shown below.

```
#!/bin/bash
export STDGPU_MAX_NDEV=1
export STDGPU_LOCK=31415
mpirun -x STDGPU_MAX_NDEV -x STDGPU_LOCK -np 2 ./mpi_lock_example.x
rm -f /dev/shm/stdcl_ctx_lock*.31415
```

Notice that the last line in the script removes a file from /dev/shm (Linux shared memory). If your application crashes it may be necessary to remove the /dev/shm lock in order to re-run your job successfully.

5.4 clvector - a C++ container using OpenCL device-sharable memory

The SDTCL memory allocator clmalloc() can be combined with C++ container classes to allow for simple object-oriented data management on the host with seamless data transfer to OpenCL devices, e.g., GPU co-processors. The clvector container class inherits directly from STL vector, .i.e., it is not an attempt to recreate a vector class. Since the STL vector class can be templated to a user-defined memory allocator, the advantages of STDCL clmalloc() can be seen as compared to the OpenCL (micro) management of memory using membuffers. By design, clmalloc() follows the regular C semantics of memory allocation and can be used as a standard memory allocator. In addition to using a memory allocator for device-sharable memory, clvector adds a few methods to STL vector that may be used to enable the complete control over memory consistency provided by OpenCL. These additional methods are necessary because a distributed memory model was never envisioned when STL vector was designed.

The following example demonstrates the use of the clvector container class in a very simple example that calculates the outer product of two vectors. The example is not intended to demonstrate any performance advantage for executing this operation on a GPU, but rather is intended to demonstrate the use of the container in a very simple context.

The OpenCL kernel code is copied here for convenience, but remains the same as that used in previous examples.

```
/* outerprod.cl */

__kernel void outerprod_kern(
   __global float* a,
   __global float* b,
   __global float* c
)
{
   int i = get_global_id(0);
   c[i] = a[i] * b[i];
}
```

The host code below shows the use of the clvector class in a simple example.

```
// clvector_example.cpp

#include <stdio.h>
#include <stdcl.h>
#include <clvector.h>

int main()
{
   size_t n = 1024;

   // use default contexts, if no GPU use CPU
   CLCONTEXT* cp = (stdgpu)? stdgpu : stdcpu;
   unsigned int devnum = 0;

   void* clh = clopen(cp,"outerprod.cl",CLLD_NOW);
   cl_kernel krn = clsym(cp,clh,"outerprod_kern",0);

   // allocate vectors using clvector
   clvector<float> a,b,c;

   // initialize vectors a[] and b[], zero c[]
   for(int i=0;i<n;i++) a.push_back(1.1f*i);
   for(int i=0;i<n;i++) b.push_back(2.2f*i);
   for(int i=0;i<n;i++) c.push_back(0.0f);

   // attach the vectors to the STDCL context
   a.clmattach(cp);
   b.clmattach(cp);
   c.clmattach(cp);

   // define the computational domain and workgroup size
   clndrange_t ndr = clndrange_init1d( 0, n, 64);

   // non-blocking sync vectors a and b to device memory (copy to GPU)
   a.clmsync(cp,devnum,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   b.clmsync(cp,devnum,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

   // set the kernel arguments
   a.clarg_set_global(cp,krn,0);
   b.clarg_set_global(cp,krn,1);
   c.clarg_set_global(cp,krn,2);

   // non-blocking fork of the OpenCL kernel to execute on the GPU
   clfork(cp,devnum,krn,&ndr,CL_EVENT_NOWAIT);

   // non-blocking sync vector c to host memory (copy back to host)
   c.clmsync(cp,0,CL_MEM_HOST|CL_EVENT_NOWAIT);
```

```
   // force execution of operations in command queue, non-blocking call
   clflush(cp,devnum,0);

   // block on completion of all operations in the command queue
   clwait(cp,devnum,CL_ALL_EVENT);

   for(int i=0;i<n;i++) printf("%f %f %f\n",a[i],b[i],c[i]);


   ////////////////////////////////////////////////////////
   ///// now resize the vectors by adding some more values ...
   ////////////////////////////////////////////////////////

   // OPTIONAL: for better performance, detach vectors from STDCL context
   a.clmdetach();
   b.clmdetach();
   c.clmdetach();

   // increase size of vectors ten-fold
   // ... note that *all* STL vector operations are valid
   for(int i=n;i<n*10;i++) a.push_back(6.6f*i);
   for(int i=n;i<n*10;i++) b.push_back(7.7f*i);
   for(int i=n;i<n*10;i++) c.push_back(0.0f);

   // OPTIONAL: ... and if you dettached the vectors, you must re-attach the
   a.clmattach(cp);
   b.clmattach(cp);
   c.clmattach(cp);

   // now follow same steps used above to sync memory, execute kernel, etc.

   a.clmsync(cp,devnum,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   b.clmsync(cp,devnum,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

   clndrange_t ndr_tenfold = clndrange_init1d( 0, n*10, 64);

   a.clarg_set_global(cp,krn,0);
   b.clarg_set_global(cp,krn,1);
   c.clarg_set_global(cp,krn,2);

   clfork(cp,devnum,krn,&ndr_tenfold,CL_EVENT_NOWAIT);

   c.clmsync(cp,0,CL_MEM_HOST|CL_EVENT_NOWAIT);

   clflush(cp,devnum,0);

   clwait(cp,devnum,CL_ALL_EVENT);

   for(int i=0;i<n*10;i++) printf("%f %f %f\n",a[i],b[i],c[i]);

   clclose(cp,clh);
}
```

5.5 clmulti_array - another C++ container using OpenCL device-sharable memory

As another example of a C++ container class using OpenCL device-sharable memory, boost::multi_array is used to create clmulti_array. This container inherits from the boost class and thus provides all of its functionality with the addition of using device-sharable memory for OpenCL devices. in the example code, a matrix-vector multiplication is carried out on the GPU where the data structures are manipulated on the host as data structures equivalent to 1D and 2D boost multi_arrays.

The kernel used here is the matrix-vector multiply kernel used in the Hello STDCL program, copied here for convenience.

```
/* matvecmult.cl */

__kernel void matvecmult_kern(
   uint n,
   __global float* aa,
   __global float* b,
   __global float* c
)
{
   int i = get_global_id(0);
   int j;
   float tmp = 0.0f;
```

```
   for(j=0;j<n;j++) tmp += aa[i*n+j] * b[j];
   c[i] = tmp;
}
```

The host code demonstrates how the containers can be manipulated on the host using the functionality of boost multi_array, while also providing the input and output data structures for the OpenCL kernels.

```
// clmulti_array_example.cpp

#include <stdio.h>
#include <stdcl.h>
#include <clmulti_array.h>

int main()
{
   cl_uint n = 1024;

   // use default contexts, if no GPU use CPU
   CLCONTEXT* cp = (stdgpu)? stdgpu : stdcpu;
   unsigned int devnum = 0;

   void* clh = clopen(cp,"matvecmult.cl",CLLD_NOW);
   cl_kernel krn = clsym(cp,clh,"matvecmult_kern",0);

   // allocate matrix and vectors using clmulti_array
   typedef clmulti_array<cl_float,1> array1_t;
   typedef clmulti_array<cl_float,2> array2_t;
   array2_t aa(boost::extents[n][n]);
   array1_t b(boost::extents[n]);
   array1_t c(boost::extents[n]);

   // initialize matrix a[] and vector b[], zero c[]
   for(int i=0;i<n;i++) for(int j=0;j<n;j++) aa[i][j] = 1.1f*i*j;
   for(int i=0;i<n;i++) b[i] = 2.2f*i;
   for(int i=0;i<n;i++) c[i] = 0.0f;

   // attach the vectors to the STDCL context
   aa.clmattach(cp);
   b.clmattach(cp);
   c.clmattach(cp);

   // define the computational domain and workgroup size
   clndrange_t ndr = clndrange_init1d( 0, n, 64);

   // non-blocking sync vectors a and b to device memory (copy to GPU)
   aa.clmsync(cp,devnum,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   b.clmsync(cp,devnum,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

   // set the kernel arguments
   clarg_set(cp,krn,0,n);
   aa.clarg_set_global(cp,krn,1);
   b.clarg_set_global(cp,krn,2);
   c.clarg_set_global(cp,krn,3);

   // non-blocking fork of the OpenCL kernel to execute on the GPU
   clfork(cp,devnum,krn,&ndr,CL_EVENT_NOWAIT);

   // non-blocking sync vector c to host memory (copy back to host)
   c.clmsync(cp,0,CL_MEM_HOST|CL_EVENT_NOWAIT);

   // force execution of operations in command queue, non-blocking call
   clflush(cp,devnum,0);

   // block on completion of all operations in the command queue
   clwait(cp,devnum,CL_ALL_EVENT);

   for(int i=0;i<n;i++) printf("%f %f\n",b[i],c[i]);


   ////////////////////////////////////////////////////////
   ///// now resize the vectors by adding some more values ...
   ////////////////////////////////////////////////////////

   n *= 3;

   // OPTIONAL: for better performance, detach containers from STDCL context
   aa.clmdetach();
```

```
    b.clmdetach();
    c.clmdetach();

    // increase size of vectors three-fold
    // ... note that *all* boost multi_array operations are valid
    aa.resize(boost::extents[n][n]);
    b.resize(boost::extents[n]);
    c.resize(boost::extents[n]);
    for(int i=0;i<n;i++) for(int j=0;j<n;j++) aa[i][j] = 1.1f*i*j;
    for(int i=0;i<n;i++) b[i] = 2.2f*i;
    for(int i=0;i<n;i++) c[i] = 0.0f;


    // OPTIONAL: ... if you dettached the containers, you must re-attach them
    aa.clmattach(cp);
    b.clmattach(cp);
    c.clmattach(cp);


    // now follow same steps used above to sync memory, execute kernel, etc.

    aa.clmsync(cp,devnum,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
    b.clmsync(cp,devnum,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

    clndrange_t ndr_threefold = clndrange_init1d( 0, n, 64);

    clarg_set(cp,krn,0,n);
    aa.clarg_set_global(cp,krn,1);
    b.clarg_set_global(cp,krn,2);
    c.clarg_set_global(cp,krn,3);

    clfork(cp,devnum,krn,&ndr_threefold,CL_EVENT_NOWAIT);

    c.clmsync(cp,0,CL_MEM_HOST|CL_EVENT_NOWAIT);

    clflush(cp,devnum,0);

    clwait(cp,devnum,CL_ALL_EVENT);

    for(int i=0;i<n;i++) printf("%f %f\n",b[i],c[i]);

    clclose(cp,clh);
}
```

5.6 clvector and CLETE - GPU acceleration with little or no effort

Notwithstanding vendor promotional material, GPU acceleration has remained difficult for average programmers and required learning alternative programming languages such as CUDA or OpenCL and re-working their code to introduce a programming model completely foreign to conventional C or C++. This reality hardly seems fair since it keeps GPU acceleration out of reach from the common programmer with no desire to exert any significant effort to rework their code.

The example below combines CLETE (Compute layer Expresion Template Engine) with the clvector contain class described above to enable GPU acceleration with virtually no effort. The single burden on the programmer is that they must include a #define prior to including the clvector.h header. By defining the macro __CLVECTOR_FULLAUTO, C++ magic happens of the kind that only expression-templating can achieve. In the spirit of this example, being targeted toward programmers who really do not care how one accelerates code using a GPU, exactly how this works will not be explained here. (Its actually quite complicated.) All that will be described is the result. When the code below is compiled, it will be automatically instrumented and when run, it will automatically generate an OpenCL kernels and the computation inside the inner loop will be perofrmed on the GPU, which is assumed to be available. What may at first glance appear to be a hack is actually quite robust, e.g., the expressions tha can be evaluated may be of arbitrary size and contain any valid C++ operations. The usefuless of the technique is presently limited to pure SIMD operations on the elements of the containers. (This might be extended in the future.) The example below is self-contained. There is no corresponding OpenCL kernel code since none is required of the programmer. For the curious, you can see the auto-generated kernel code by setting the environment variable COPRTHR_LOG_AUTOKERN, in which case it will be written out to a file in the run directory.

```
// clete_clvector_example.cpp

#include <iostream>
using namespace std;

#include "Timer.h"

////////////////////////////////////////////////////////////////////////////
// #define __CLVECTOR_FULLAUTO to enable CLETE automatic GPU acceleration
// for pure SIMD operations on clvector data objects.
//
// Set the environmaent variable COPRTHR_LOG_AUTOKERN to see the automatically
```

```
// generated OpenCL kernels used to execute the computation on GPU.
//
// With the #define commented out standard expression-templating is used
// to efficiently execute the computation on the CPU.
//////////////////////////////////////////////////////////////////////////////

#include <stdcl.h>
//#define __CLVECTOR_FULLAUTO
#include <clvector.h>

int main()
{
   Setup(0);
   Reset(0);

   int n = 1048576;

   clvector<float> a,b,c,d,e;

   for (int i=0; i<n; ++i) {
      a.push_back(1.1f*i);
      b.push_back(2.2f*i);
      c.push_back(3.3f*i);
      d.push_back(1.0f*i);
      e.push_back(0.0f);
   }

   Start(0);
   for(int iter=0; iter<10; iter++) {
      e = a + b + 2112.0f * b + sqrt(d) - a*b*c*d + c*sqrt(a) + a*cos(c);
      a = a + log(fabs(e));
   }
   Stop(0);
   double t = GetElapsedTime(0);

   for (int i=n-10; i<n; ++i) {
      cout << " a(" << i << ") = " << a[i]
           << " b(" << i << ") = " << b[i]
           << " c(" << i << ") = " << c[i]
           << " d(" << i << ") = " << d[i]
           << " e(" << i << ") = " << e[i]
           << endl;
   }

   printf("compute time %f (sec)\n",t);

}
```

5.7 clmulti_array and CLETE - another example of automatic GPU acceleration

The example below is identical to the previous section, but in this case demonstrates that the same C++ magic trick can be peformed using clmulti_array. For help with the syntax, see the boost::multi_array documentation since clmulti_array inherits from this class and provides a superset of functionality.

```
// clete_clmulti_array_example.cpp

#include <iostream>
using namespace std;

#include "Timer.h"

//////////////////////////////////////////////////////////////////////////////
// #define __CLMULTI_ARRAY_FULLAUTO to enable CLETE automatic GPU acceleration
// for pure SIMD operations on clvector data objects.
//
// Set the environmaent variable COPRTHR_LOG_AUTOKERN to see the automatically
// generated OpenCL kernels used to execute the computation on GPU.
//
// With the #define commented out standard expression-templating is used
// to efficiently execute the computation on the CPU.
//////////////////////////////////////////////////////////////////////////////

#include <stdcl.h>
#define __CLMULTI_ARRAY_FULLAUTO
#include <clmulti_array.h>
```

```
int main()
{
   Setup(0);
   Reset(0);

   typedef clmulti_array< float, 1 > array1_t;
   typedef clmulti_array< float, 2 > array2_t;
   typedef clmulti_array< float, 3 > array3_t;
   typedef clmulti_array< float, 4 > array4_t;

   array1_t a(boost::extents[100]);
   array2_t b(boost::extents[100][30]);
   array3_t c(boost::extents[100][30][45]);
   array4_t d(boost::extents[100][30][45][60]);
   array4_t x(boost::extents[100][30][45][60]);

   for(int i = 0; i<100; i++) {
      a[i] = i;
      for(int j=0; j<30; j++) {
         b[i][j] = i*j;
         for(int k=0; k<45; k++) {
            c[i][j][k] = i+j+k;
            for(int l=0; l<60; l++) d[i][j][k][l] = i*j*k*l;
         }
      }
   }

   Start(0);
   for(int iter=0;iter<10;iter++) {
      x = a*b*c*d + sqrt(d) -81.0f + pow(c*d,0.33f);
      for(int i = 0; i<100; i++) a[i] = cos(x[i][0][0][0]);
   }
   Stop(0);
   double t = GetElapsedTime(0);

   for(int i=0;i<10;i++) cout<<i<<" "<<x[i][i][i][i]<<endl;

   cout<<"compute time "<<t<<" (sec)\n";

}
```

5.8 clcontext_info_example

One nice feature of STDCL is that it provides default contexts that are ready to use by the programmer. In some cases, it might be interesting or useful to examine exactly what is contained in a give context. The following example exercises some utilty routines that can be used to query a CL context for a description of what they contain.

```
/* clcontext_info_example.c */

#include <stdio.h>
#include <stdcl.h>

int main()
{
   CLCONTEXT* cp;

   cp = stddev;

   if (cp) {
      printf("\n***** stddev:\n");
      int ndev = clgetndev(cp);

      struct clstat_info stat_info;
      clstat(cp,&stat_info);

      struct cldev_info* dev_info
         = (struct cldev_info*)malloc(ndev*sizeof(struct cldev_info));
      clgetdevinfo(cp,dev_info);

      clfreport_devinfo(stdout,ndev,dev_info);

      if (dev_info) free(dev_info);
   }
```

```
   cp = stdcpu;

   if (cp) {
      printf("\n***** stdcpu:\n");
      int ndev = clgetndev(cp);

      struct clstat_info stat_info;
      clstat(cp,&stat_info);

      struct cldev_info* dev_info
         = (struct cldev_info*)malloc(ndev*sizeof(struct cldev_info));
      clgetdevinfo(cp,dev_info);

      clfreport_devinfo(stdout,ndev,dev_info);

      if (dev_info) free(dev_info);
   }


   cp = stdgpu;

   if (cp) {
      printf("\n***** stdgpu:\n");
      int ndev = clgetndev(cp);

      struct clstat_info stat_info;
      clstat(cp,&stat_info);

      struct cldev_info* dev_info
         = (struct cldev_info*)malloc(ndev*sizeof(struct cldev_info));
      clgetdevinfo(cp,dev_info);

      clfreport_devinfo(stdout,ndev,dev_info);

      if (dev_info) free(dev_info);
   }

}
```

5.9 bdt_nbody and bdt_em3d

The COPRTHR SDK example/ directory also contains two demo applications - bdt_nbody and bdt_em3d. The N-body demo (bdt_nbody) is very similar to the BDT NBody Tutorial, however, the source code is a bit more complex since it includes an OpenCL display and the kernel is optimized for performance. The 3D FDTD electromegnetic demo (bdt_em3d) also provides an OpenGL display. Note that due to the interaction with OpenGL, these examples sometimes have difficulty working properly. The issue is normally a problem with the installed OpenGL utility libraries.

---

8. Tools

8.1 `clld`: how to create single executables with embedded OpenCL kernel code

When developing and debugging OpenCL kernel code, the use of OpenCL just-in-time (JIT) compilation can be quite convenient. Making a small change to kernel code does not require that the application be recompiled, but rather simply re-run. However, when distributing software to others, requiring users to drag .cl files around with the executable is cumbersome and prone to errors. The solution for most programmers will be to incorporate the kernel code as strings into the application, and after doing this several times, they will eventually come up with their own scheme for wrapping and simplifying things to shield themselves from this most annoying aspect of OpenCL. STDCL provides an elegant solution to this problem with the addition of the tool `clld`.

`clld` allows OpenCL kernels to be embedded into an ELF object and linked into the executable using a standard linker such as `ld`. The kernel source or binary code is stored in separate sections within the ELF file in manner completely transparent to those sections used for representing the program itself. The operation of `clld` is integrated with the run-time loader provided by STDCL in such a way that the OpenCL kernels can be built and compiled transparently with almost no effort by the programmer. The simplicity is best illustrated by example. Consider a simple program that doubles the values of a vector using a GPU. The kernel code might look like this, stored in the file foo.cl:

```
/* foo.cl */

__kernel void
foo_kern( __global float* a )
{
   int gtid = get_global_id(0);
   a[gtid] *= a[gtid];
}
```

The host program might look like this, stored in the file foo.c:

```
/* foo.c */
#include <stdio.h>
#include <stdcl.h>
in main()
{
        void* clh = clopen(stdgpu,0,CLLD_NOW);
        cl_kernel krn = clsym(stdgpu,clh,"foo_kern",0);
        float* aa = (float*)clmalloc(stdgpu,1000*sizeof(float));
        int i;
        for(i=0;i<1000;i++) aa[i] = 1.1f*i;
        clndrange_t ndr = clndrange_init1d( 0, 1000, 64);
        clmsync(stdgpu,0,aa,CL_MEM_DEVICE|CL_EVENT_WAIT);
        clfork(stdgpu,0,krn,&ndr,Cl_EVENT_WAIT);
        clmsync(stdgpu,0,aa,CL_MEM_HOST|CL_EVENT_WAIT);
        clfree(aa);
}
```

Note that the second argument to `clopen()` is `null` - this is where one normally specifies the `.cl` file that contains the kernel source. Setting this argument to `null` instead of a filename tells the dynamic loader to return a handle to the kernel source embedded within the executable. In order for this to work, the necessary kernel source would need to have been linked into the executable, which can be done with the following steps:

```
clld --cl-source foo.cl
gcc -o foo.x foo.c out_clld.o
```

Note that `clld` is not limited to embedding a single `.cl` but rather can be used to embed the kernel source from multiple files listed on the cammand line. The default output of clld is the ELF object file `out_clld.o`. The way in which clld embeds the kernel source into the ELF object can be seen by inspecting the final executable using readelf command which will show that there are 5 additional sections not normally found within a tyical ELF object or executable. Specifically, the sections, `.clprgs, .cltexts, .clprgb, .cltextb,` and `.clstrtab` are used to embed the kernel code.

6.2 cltrace: tracing OpenCL calls

The tool cltrace can be used to trace OpenCL calls on the host and is modeled after the UNIX command strace. No instrumentation of the application is necessary and cltrace may be used with any compilant OpenCL implementation. As an example, running the Hello STDCL example using cltrace produces the following output. Tracing OpenCL calls can be useful since most, but not all, OpenCL calls return 0 on success and a negative integer on error. Thus, following the trace of an application and spotting the first sign of an error code can useful during the debugging process.

If the option -T is used as the first command line argument to cltrace then timing information will be reported for each call. The time is reported in seconds appended to the call and enclosed within < > brackets. Additional options are not supported at this time.

```
# [1]cltrace ./hello_stdcl.x
cltrace: using OpenCL lib '/usr/local/atistream//lib/x86_64/libOpenCL.so.1'
cltrace1: clGetPlatformIDs(0,(nil),1) = 0
cltrace1: clGetPlatformIDs(1,0x153ad3b0,1) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,13,0x153ad160,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,37,0x153ad380,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,36,0x153ad110,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,29,0x15704e10,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,56,0x153ad180,(nil)) = 0
cltrace1: clCreateContextFromType(0x153ad140,0xffffffffffffffff,(nil),(nil),0) = 356176448
cltrace1: clGetContextInfo(0x153ad240,0x1081,0,(nil),0x7fffff9c6028) = 0
cltrace1: clGetContextInfo(0x153ad240,0x1081,24,0x153ae0a0,(nil)) = 0
cltrace1: clCreateCommandQueue(0x153ad240,0x156facf0,(nil),0) = 356180432
cltrace1: clCreateCommandQueue(0x153ad240,0x156fe070,(nil),0) = 367828960
cltrace1: clCreateCommandQueue(0x153ad240,0x15703c00,(nil),0) = 367558368
cltrace1: clGetPlatformIDs(0,(nil),1) = 0
cltrace1: clGetPlatformIDs(1,0x16155b30,1) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,13,0x161b4730,(nil)) = 0
```

```
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,37,0x161d8530,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,36,0x161b9b70,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,29,0x161ac0e0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,56,0x15ab16e0,(nil)) = 0
cltrace1: clCreateContextFromType(0x161abd10,0x2,(nil),(nil),0) = 371170096
cltrace1: clGetContextInfo(0x161f9b30,0x1081,0,(nil),0x7fffff9c6028) = 0
cltrace1: clGetContextInfo(0x161f9b30,0x1081,8,0x161b0950,(nil)) = 0
cltrace1: clCreateCommandQueue(0x161f9b30,0x15703c00,(nil),0) = 370998688
cltrace1: clGetPlatformIDs(0,(nil),1) = 0
cltrace1: clGetPlatformIDs(1,0x161fa410,1) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,1024,0x7fffff9c5ab0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x900,13,0x153ad3b0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x901,37,0x1614f9d0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x902,36,0x15e82100,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x903,29,0x161f8fd0,(nil)) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,0,(nil),0x7fffff9c6018) = 0
cltrace1: clGetPlatformInfo(0x2b2776434800,0x904,56,0x15e51ef0,(nil)) = 0
cltrace1: clCreateContextFromType(0x15ab1760,0x4,(nil),(nil),0) = 367347104
cltrace1: clGetContextInfo(0x15e545a0,0x1081,0,(nil),0x7fffff9c6028) = 0
cltrace1: clGetContextInfo(0x15e545a0,0x1081,16,0x161606d0,(nil)) = 0
cltrace1: clCreateCommandQueue(0x15e545a0,0x156facf0,(nil),0) = 367337680
cltrace1: clCreateCommandQueue(0x15e545a0,0x156fe070,(nil),0) = 374568864
cltrace1: clUnloadCompiler() = 0
cltrace1: clCreateProgramWithSource(0x15e545a0,1,0x7fffff9c61c0,0x7fffff9c61b8,0) = 371900800
cltrace1: clBuildProgram(0x162ac180,2,0x161606d0,0x7fffff9c5d60,(nil),(nil)) = 0
cltrace1: clGetProgramBuildInfo(0x162ac180,0x156facf0,0x1183,0,(nil),0x7fffff9c6168) = 0
cltrace1: clGetProgramBuildInfo(0x162ac180,0x156facf0,0x1183,1,0x161fa410,(nil)) = 0
cltrace1: clCreateKernelsInProgram(0x162ac180,0,(nil),0x162a7750) = 0
cltrace1: clCreateKernelsInProgram(0x162ac180,1,0x161fa410,(nil)) = 0
cltrace1: clGetKernelInfo(0x162f8c60,0x1190,256,0x162cb350,(nil)) = 0
cltrace1: clGetKernelInfo(0x162f8c60,0x1191,4,0x162cb450,(nil)) = 0
cltrace1: clGetKernelInfo(0x162f8c60,0x1192,4,0x162cb454,(nil)) = 0
cltrace1: clGetKernelInfo(0x162f8c60,0x1193,8,0x162cb458,(nil)) = 0
cltrace1: clGetKernelInfo(0x162f8c60,0x1194,8,0x162cb460,(nil)) = 0
cltrace1: clCreateBuffer(0x15e545a0,33,16384,0x163f8e80,0) = 372787920
cltrace1: clCreateBuffer(0x15e545a0,33,256,0x162f4900,0) = 373280416
cltrace1: clCreateBuffer(0x15e545a0,33,256,0x162d0220,0) = 371900224
cltrace1: clEnqueueWriteBuffer(0x15e520d0,0x16384ad0,0,0,16384,0x163f8e80,0,(nil),0x7fffff9c6278) = 0
cltrace1: clEnqueueWriteBuffer(0x15e520d0,0x163fcea0,0,0,256,0x162f4900,0,(nil),0x7fffff9c6278) = 0
cltrace1: clSetKernelArg(0x162f8c60,0,4,0x7fffff9c633c) = 0
cltrace1: clSetKernelArg(0x162f8c60,1,8,0x163f8e30) = 0
cltrace1: clSetKernelArg(0x162f8c60,2,8,0x162f48b0) = 0
cltrace1: clSetKernelArg(0x162f8c60,3,8,0x162d01d0) = 0
cltrace1: clEnqueueNDRangeKernel(0x15e520d0,0x162f8c60,1,0x7fffff9c62d8,0x7fffff9c62f8,0x7fffff9c6318,0,(nil),0x7fffff9c6290) = 0
cltrace1: clEnqueueReadBuffer(0x15e520d0,0x162abf40,0,0,256,0x162d0220,0,(nil),0x7fffff9c6278) = 0
cltrace1: clFlush(0x15e520d0) = 0
cltrace1: clWaitForEvents(1,0x16848710) = 0
cltrace1: clReleaseEvent(0x162fc1f0) = 0
cltrace1: clWaitForEvents(3,0x16696770) = 0
cltrace1: clReleaseEvent(0x162865f0) = 0
cltrace1: clReleaseEvent(0x16256aa0) = 0
cltrace1: clReleaseEvent(0x1625bb90) = 0
0 0.000000 0.000000
1 2.200000 206532.468750
2 4.400000 413064.937500
 ...
63 138.600006 13011548.000000
cltrace1: clReleaseMemObject(0x16384ad0) = 0
cltrace1: clReleaseMemObject(0x163fcea0) = 0
cltrace1: clReleaseMemObject(0x162abf40) = 0
cltrace1: clReleaseKernel(0x162f8c60) = 0
cltrace1: clReleaseProgram(0x162ac180) = 0
# [2]
```

7 Known Issues

- Exceeding the maximum event list size of 1024 with kernel and/or memory events, without blocking for completion, will cause events to be dropped and undefined behavior. This is not a bug since failing to drain a command queue with more than 1024 operations pending is poor programming.
- Calls to `clglmalloc()` may fail due to a known bug in certain OpenCL SDKs that return an incorrect value for the GL buffer size.
- A default context, e.g., `stdgpu`, should be tested to ensure it is not `0` before use. This is a common cause of seg faults and the root problem is that the underlying OpenCL SDK has found no valid devices to run on. A typical cause is failing to select the correct platform for GPU support.
- For platforms with multiple devices, clflush() may not initiate kernel execution so as to allow for concurrent computation as it is intended. This is due to a known flaw in one of the vendor OpenCL implementations which prevents concurency and not a flaw in the implementation of STDCL.
- The Windows 7 support is presently Beta support only. If difficulties are encountered, please, send a note to support@browndeertechnology.com .

---

revised 18 June 2011 by DAR