Brown Deer Technology

The COPRTHR® Primer rev. 1.6

Contents

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

---

1 Overview of Libraries and Tools

The CO-PRocessing THReads (COPRTHR) SDK provides libraries and tools for developing applications that target the multi-threaded co-processing parallelism of modern multi-core and many-core processors. This includes heterogeneous CPU/GPU hybrid systems and other emerging processor architectures. The COPRTHR SDK leverages upon OpenCL and may be used to develop OpenCL enabled applications. The range of hardware supported by the SDK is limited only by the availability of a suitable OpenCL implementation, and includes support for processors from AMD, Intel, and Nvidia. For architectures and operating systems for which a vendor implementation of OpenCL is not available, COPRTHR provides an open-source OpenCL implementation for multi-core x86, ARM and Epiphany processors. The primary operating system supported by the SDK is Linux. However, limited support is provided for Windows 7 and FreeBSD-8. The software stack for COPRTHR is shown in the figure below along with additional 3rd-party software that the SDK supports.

1.1 STDCL

The STandarD Compte Layer (STDCL) is a powerful API built on top of OpenCL that is much easier to use than OpenCL itself. STDCL (STandarD Compute Layer) has been developed as an API for OpenCL applications in order to simplify application development and introduce more intuitive programming semantics. STDCL is designed in a style inspired by conventional UNIX APIs for C programming, and may be used directly in C/C++ applications, with additional support for Fortran through direct API bindings.

The STDCL API provides support for default compute contexts, an integrated dynamic CL program loader, memory management, kernel management, and scheduling for asynchronous operations. Environment variables provide run-time control over certain aspects of the API including the prioritized selection of platforms and

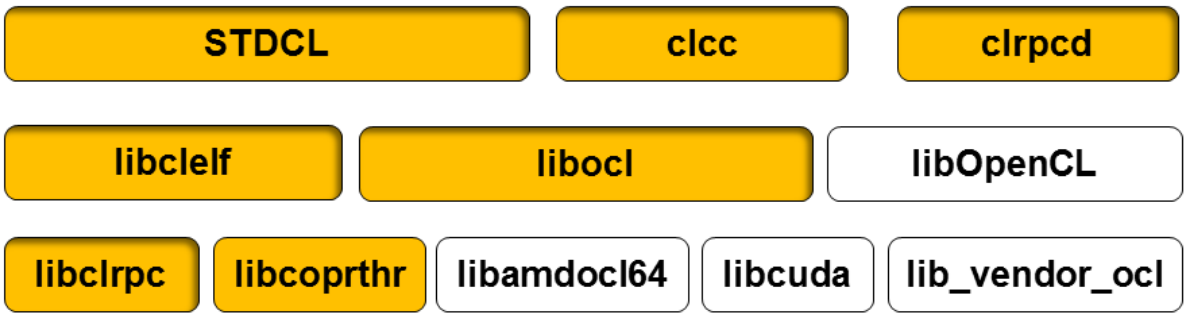

Figure 1: COPRTHR SDK Software Stack

the management of co-processing resources across multiple processes. This support is especially useful in providing transparent support for MPI-based parallelism on multi-GPU systems.

STDCL replaces tedious low-level OpenCL host calls with semantics that better address the real programming requirements for application developers. At the same time, the API does not inhibit direct access to OpenCL host calls and data structures for the infrequent cases where they are actually needed.

1.2 clcc

The SDK provides an offline compiler and linker for multi-vendor multi-device cross-compilation that employs a real compilation model. This is achieved by defining extensions to the standard ELF sections of an ordinary object file or executable capable of representing OpenCL kernel programs in source and binary form. These sections are capable of supporting the full range of cross-compilation that a programmer will encounter when developing applications designed to utilize co-processing devices. The offline compiler and linker are designed to behave the way a programmer would expect them to behave and fit seamlessly into the normal GCC-type workflow employed by serious programmers developing serious applications.

1.3 libclelf

The ELF extensions for OpenCL (CL-ELF) are formally implemented withing the library libclelf which is use by the offline compiler and linker to create cross-compiled linkable object files. The dynamic loader that is provided as part of the STDCL implementation uses this same library to load the correct source or binary kernel for use within an application.

1.4 libocl

The SDK includes an OpenCL loader that is backward compatible with the conventional libOpenCL ICD loader. The purpose of libocl.so, which may be aliased to libOpenCL.so, is to provide a much greater set of capabilities than what is provided with the standard loader. Among these capabilities are more sensible system configuraton and resource management options, as well as the creation of custom contexts based on the hardware that is actually installed on a given system. OpenCL platform configuration issues are more reliably addressed by the user or system administrator, and not the application developer. Additionally, libocl provides extensible hooks and accounting support, as well as direct support for CLRPC.

1.5 libcoprthr

The SDK includes an open source OpenCL implementation for x86, ARM and Epiphany multi-core processors. This implementation can be useful on architectures and operating systems for which no vendor provided OpenCL implementation is available. Additionally, this implementation has proven to yield faster benchmarks than vendor provided x86 implementions on certain real-world HPC benchmarks.

1.6 libclrpc/clrpcd

The SDK includes a Compute Layer Remote Procedure Call (CLRPC) implementation that consists of a client-side RPC OpenCL implementation, libclrpc, and a CLRPC server, clrpcd, that is used to export local OpenCL implementations over a network. CLRPC allows OpenCL host applications to access networked compute devices.

2 Test Scripts

For Linux and FreeBSD installatons, a set of test scripts are provided as a quick check to verify the installation. (For detailed instructions describing the installation of the COPRTHR SDK, see the release notes for the specific version being installed.) 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 test can be re-run to provide debug information by executing:

```
] cldebug -- ./test_<name>.x --size 65536 --blocksize 16
```

where ./test_.x refers to the specific test that failed.

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

Please note that the code used in these tests is *absolutely not* a useful way to learn how to use SDK. The code is generated by scripts and will be relatively unintelligible. The examples/ directory is the best place to get started learning about the SDK and how it simplifies the use of OpenCL.

3 libocl: An OpenCL Loader with Precision and Functionality

3.1 A Replacement for the standard libOpenCL Loader

The 'libocl' library is a backward compatible replacement for the standard Khronos OpenCL loader, and replaces the ICD platform enumeration with a more precise and flexible mechanism for defining platform policy and configuration. In addition, 'libocl' provides features not found with standard OpenCL loaders.

In order to use 'libocl', the user can either modify the library that applications are linked against by simply changing '-lOpenCL' to '-locl'. Alternatively, a soft link can be used to alias the library to the one that a precompiled application expects to find, i.e.,

```
link -s libocl.so libOpenCL.so
```

With this release, the libocl loader should support applications developed with the OpenCL 1.1 specification.

3.2 Precise Platform Configuration: `ocl.conf` files

The 'libocl' loader replaces the random enumeration of '.icd' files placed in the '/etc/OpenCL/vendors/' directory with a precise configuration file that may be set by the system admin and then overriden by any user following a well-defined ordering of search paths.

The following order for search paths provides for an increasingly specialized control over the OpenCL platform configuration that a given application sees.

```
./ocl.conf
./.ocl.conf
$HOME/ocl.conf
$HOME/.ocl.conf
/etc/ocl.conf
```

The syntax of an 'ocl.conf' file is based on 'libconfig' which is used for parsing the content. This syntax provides a compact hierarchical structure with C-like constructs. For convenience ordinary C and shell syntax for comments is respected, allowing the configuration file to be quickly modified to enable/disable a given line or section.

The primary section of an 'ocl.conf' file is the *platforms* section, and is best illustrated by example.

```
platforms = (
    { platform="coprthr"; lib="libcoprthr.so"; },
#   { platform="nvidia"; lib="libcuda.so"; },
    { platform="intel"; lib="libintelocl.so"; }
);
```

In this example, an application would see the OpenCL platforms *coprthr* and *intel*, but the *nvidia* platform would not be seen since it is commented out. In this way a system administrator or user can control the OpenCL platforms presented to an application and not rely upon selection mechanisms, if any, within an application. This control replaces the technique of "hiding" `.icd` files in `/etc/OpenCL/vendors/`, an option only open to users with root permission.

If for whatever reason, the `libocl` loader cannot find an `ocl.file` on the system, it will then use the `.icd` files found in `/etc/OpenCL/vendors/`. As an additional convenience, an ordered list of directories to search for `.icd` files may be specified. This resolves a common challenge faced by users without root permission, being unable to modify files in the `/etc/` directory. The syntax for specifying alternate ICD directories is,

```
icd_dirs = (
    "/etc/OpenCL/vendors/",
    "/home/user/local/etc/OpenCL/vendors",
    "/home/user/icd"
);
```

3.3 Remote CLRPC Servers

The `ocl.conf` file can be used to add platforms that have been exported over a network using CLRPC servers. A detailed discusion of CLRPC is provided in the section on CLRPC: OpenCL Remote Procedure Calls. An example of a *clrpc* section in an `ocl.conf` file is shown below

```
clrpc = {
    enable = "yes";
    servers = (
        { url="127.0.0.1:2112" },
        { url="192.168.1.15:2112" },
#       { url="192.168.1.16:2112" },
        { url="10.1.2.3:2112" },
        { url="10.1.2.3:2113" },
        { url="10.1.2.3:2114" }
    );
};
```

The CLRPC servers will be checked for platforms by contacting a CLRPC server listening at the specified address and port. If no response is initially received when contacted, the server is simply skipped.

3.4 Hooks and Accounting

At present some functionality of the `libocl` loader will remain hidden to the user, but it may still be of interest to understand what is going on behind the scenes. The OpenCL ICD specification for the loader defines a call vector mechanism that forwards OpenCL host calls to the correct vendor platform implementation. Such a mechanism provides a perfect opportunity for call interception, and the `libocl` loader exploits this to enable call tracing, process accounting, and extensible pre- and post-call hooks.

This mechanism is presently used to provide the accounting information for the `cltop` command. The implementation of the `libocl` loader provides for extensible pre- and post-call hooks that can, in principle, be extended for advanced applications. These hooks will be used in a future release to support an integrated run-time debugger.

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. 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()
{
   stdcl_init(); /* requred for Windows only, Linux and FreeBSD will ignore this call */

   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). Next, fork the kernel execution on the GPU (non-blocking), passing the kernel arguments directly to the new clforka() call that will set the arguments automatically, and then 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.

New to this release is support for Fortran bindings to STDCL. Quite simply, this allows nearly all of the functionality provided by STDCL to be available to Fortran programmers, providing a simple powerful interface to OpenCL programming from Fortran applications. The example below is nearly identical to the hello STDCL example above, except it is written in Fortran. On detail to note is that the opaque OpenCL and STDCL types, which are merely pointers in practice, must be referenced directly and generically as C pointers since Fortran does not support type alising.

```
!!!! hello_stdcl.f90

program main
```

```
use iso_c_binding
use stdcl

implicit none

integer(C_INT) :: n = 64
type(C_PTR) :: cp
integer(C_INT) :: devnum = 0
type(C_PTR) :: clh
type(C_PTR) :: krn
type(C_PTR) :: p_aa, p_b, p_c
real(C_FLOAT), pointer :: aa(:,:), b(:), c(:)
integer :: i,j
type(clndrange_struct), target :: ndr
integer(C_INT) :: rc
type(C_PTR) :: ev

!!!! use default contexts, if no GPU use CPU
if (C_ASSOCIATED(stdgpu)) then
  cp = stdgpu
else
  cp = stdcpu
endif

!!!! build CL program, get kernel
clh = clopen(cp,"matvecmult.cl"//C_NULL_CHAR,CLLD_NOW);
krn = clsym(cp,clh,"matvecmult_kern"//C_NULL_CHAR,0);

!!!! allocate OpenCL device-sharable memory, associate with fortran pointers
p_aa = clmalloc(cp,int8(n*n*4),0);
call C_F_POINTER(p_aa,aa,[n,n])

p_b = clmalloc(cp,int8(n*4),0);
call C_F_POINTER(p_b,b,[n])

p_c = clmalloc(cp,int8(n*4),0);
call C_F_POINTER(p_c,c,[n])

!!!! initialize vectors a[] and b[], zero c[]
do i = 1,n
  do j = 1,n
    aa(i,j) = 1.1 * (i-1) * (j-1)
  end do
end do

do i = 1,n
  b(i) = 2.2 * (i-1)
end do

do i = 1,n
  c(i) = 0.0
end do
```

```
  !!!! define the computational domain and workgroup size
  ndr = clndrange_init1d( 0, n, 64)

  !!!! non-blocking sync vectors a and b to device memory (copy to GPU)
  ev = clmsync(cp,devnum,p_aa,CL_MEM_DEVICE+CL_EVENT_NOWAIT)
  ev = clmsync(cp,devnum,p_b,CL_MEM_DEVICE+CL_EVENT_NOWAIT)

  !!!! set the kernel arguments
  rc = clarg_set(cp,krn,0,n);
  rc = clarg_set_global(cp,krn,1,p_aa);
  rc = clarg_set_global(cp,krn,2,p_b);
  rc = clarg_set_global(cp,krn,3,p_c);

  !!!! non-blocking fork of the OpenCL kernel to execute on the GPU
  ev = clfork(cp,devnum,krn,C_LOC(ndr),CL_EVENT_NOWAIT)

  !!!! non-blocking sync vector c to host memory (copy back to host)
  ev = clmsync(cp,0,p_c,CL_MEM_HOST+CL_EVENT_NOWAIT)

  !!!! force execution of operations in command queue (non-blocking call)
  rc = clflush(cp,devnum,0)

  !!!! block on completion of operations in command queue
  ev = clwait(cp,devnum,CL_ALL_EVENT)

  do i = 1,n
    write(*,*) i,b(i),c(i)
  end do

  rc = clfree(p_aa)
  rc = clfree(p_b)
  rc = clfree(p_c)

  rc = clclose(cp,clh)

end
```

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 some of these examples will not work with Windows 7. The examples that rely on CLETE will not work since templated metaprogramming can be problematic with the MSVS interpretation of the C++ standard. However, with this release support for the C++ containers clvector and clmulti_array now work, and those examples are now provided under the msvs2010/examples/ directory.

5.1 clopen_example - Managing OpenCL Kernel Code

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 Managing OpenCL Kernel Code ELIMINATED

New to this release is a feature of the clcc compiler tools that allows strong binding of kernel symbols using a standard compilation model. This completely eliminates the management of program and kernel objects.

Given the same outer product kernel used in the previous example,

```
/* 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];
}
```

we can compile the code using clcc using the options to cause strong symbol binding in the compiled object code,

```
clcc -j -k -c outerprod.cl
```

which will produce the file `outerprod.o`. The host code is now modified as shown below.

```
/* example_strong_binding.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;

   /* 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);

   /* non-blocking fork of the OpenCL kernel to execute on the GPU */
   clforka(cp,devnum,outerprod_kern,&ndr,CL_EVENT_NOWAIT,a,b,c);

   /* 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);
}
```

Note that the clopen() and clsym() calls are eliminated and the symbol `krn` is replaced with the actual symbol from the kernel source code, `outerprod_kern`.

The host code is compiled using,

```
gcc example_strong_binding.c outerprod.o
```

and the resulting executable will simple work, no management of OpenCL programs or kernels, all of that management is eliminated.

5.3 image2d_example - Using Texture Memory for Fast Lookup Tables

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);
}
```

5.4 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.
```

5.5 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()
{
   stdcl_init(); /* requred for Windows only, Linux and FreeBSD will ignore this call */

   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);
```

```
}
```

5.6 clmulti_array - Another C++ Container Using 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()
{
   stdcl_init(); /* requred for Windows only, Linux and FreeBSD will ignore this call */

   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);
}
```

5.7 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 Expression 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 performed 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 that can be evaluated may be of arbitrary size and contain any valid C++ operations. The usefulness 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);

}
```

5.8 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";

}
```

5.9 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);
   }

}
```

5.10 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.

6 Tools

6.1 CL-ELF: A Real Compilation Model for OpenCL, Finally

By default OpenCL provides great flexibility, but offers no guidance or support for a real compilation model of the kind most programmers would expect. Instead this is left as an exercise. COPRTHR provides a set of tools and libraries that support a robust compilation model for OpenCL with sensible fallback behaviors. At the core of the compilation model is an extension to the standard ELF object format (CL-ELF) that provides sections for logically supporting all that is possible with OpenCL in terms of cross-compilation.

The tools include an offline compiler (clcc) capable of generating linkable object files containing source and binaries for any number of platforms and devices. Multiple object files can be combined using a linker (clld) to create a static object or shared library. A full range of options are provided to allow the programmer complete control over the content of each object file. Additional utilities are provided to extract information and further manipulate this extended ELF format. The STDCL dynamic loader provides integrated support for the CL-ELF format, enabling a programmer to simply link their OpenCL kernels and access them by symbol name.

The following simple examples demonstrate the use of the offline compiler to produce a single linkable ELF object file that may used to produce self-contained executables without the need for JIT compilation of auxiliary .cl files.

Example 1: Compile three kernels stored in three files to create a linkable object file and link this into an application:

```
] clcc alpha.cl beta.cl gamma.cl -o all_kernels.o
] gcc -o my_app.x my_app.c all_kernels.o
```

Example 2: Compile the three kernels separately, link them to create a single object file, and link this into an application:

```
] clcc alpha.cl
] clcc beta.cl
] clcc gamma.cl
] clld alpha.o beta.o gamma.o -o all_kernels.o
] gcc -o my_app.x my_app.c all_kernels.o
```

Example 3: Compile the three kernels separately, link them to produce a single object file, strip out the source code and only include binaries for an AMD Cayman and all Nvidia devices:

```
] clcc alpha.cl
] clcc beta.cl
] clcc gamma.cl
] clld -o all_kernels.o -mdevice=amdapp:cayman,nvidia: -b alpha.o beta.o gamma.o
] gcc -o my_app.x my_app.c all_kernels.o
```

So where is my kernel? That is the best part about the compilation model. From inside the application using the STDCL dynamic loader, kernels are accessed with a single call, for example:

```
cl_kernel krn_alpha = clsym(stdgpu,"alpha_kernel",CLLD_NOW);
```

That's it. Simple, efficient, portable and reliable. For programmers who actually enjoy the tedious and error prone steps involved with directly managing OpenCL kernel code from within their application, this compilation model is not for you.

NOTE: Object files generated by clcc and/or clld can only be combined using clld. Attempts to use the standard linker ld for this purpose will, by design, cause a link time error due to multiple hash symbol definitions. This feature was deliberately introduced since a standard linker cannot properly combine sections in the CL-ELF format yet. The standard linker ld is used once to link a single CL-ELF file with conventional ELF object files.

6.2 clcc: An Offline Compiler for OpenCL

clcc is an offline compiler for OpenCL kernel source files used to generate a linkable ELF object containing multiple binaries targeting multiple platforms and devices. The original source code can also be included for subsequent JIT compilation on targets for which no binary is included. The ouptut of clcc uses an open extension of the standard ELF format (CL-ELF) that enables the representation of the full range of cross-compilations avaialable with multiple OpenCL platform implementations. The output can be linked as many times as necessary using clld, but can only be linked once to produce an ELF object or executable file using the conventional linker ld. In order to protect against multiple links using ld, hash values are included in CL-ELF object files such that multiple values can only be resolved with clld. The following is a synopsis of the usage for clcc.

```
clcc [options] file1.cl file2.cl ... [-o output.o]
```

Options:

-b Include only binaries in the CL-ELF object, i.e., do not embed the original source code. This prevents the kernels from being JIT compiled on target platforms and devices not included in the offline compilation.

-c Generate an object file that is linkable with clld. This is the default behavior and the flag is provided only to maintain conventional compiler semantics.-fopenclSpecify the language dialect for compilation to be strict OpenCL, overriding that which is inferred from the file extensions.-fcuda(Reserved)

-fopenmp (Reserved)

-fstdcl (Reserved)

-h –help Print a brief help message.

-I Add to the include path for compilation.

-mall Include binaries for all devices supported by all available platforms.

-mavail Include binaries for only those devices available on the host system.-mdevice=Select exclusive list of devices to include where is a comma separated list with no spaces. Device names are vendor specific. Note that the naming convention will in some cases employ device aliases, e.g., all x86_64 processors are identified with that simple tag regardless of the exact processor name.

-mdevice-exclude= Select list of devices to exclude where is a comma separated list with no spaces. Device names are vendor specific. Note that the naming convention will in some cases employ device aliases, e.g., all x86_64 processors are identified with that simple tag regardless of the exact processor name.

-mplatform= Select exclusive list of platforms to include where is a comma separated list with no spaces.

-mplatform-exclude= Select list of platforms to exclude where is a comma separated list with no spaces.

-o

: Specifiy the output filename for the final ELF object file. The default naming convention for compiling a single OpenCL kernel file is the filename base with the .o extension. The default naming convention for compiling multiple OpenCL kernel files is out_clcc.o .

-s Include only source the original source code in the CL-ELF object, i.e., do not embed any device specific binaries. This will require the kernels to be JIT compiled on all target platforms and devices.

-v Generate verbose diagnostic information.

–**version** Print version information.

6.3 clld: An Offline Linker for OpenCL

clld is an offline linker used to combine CL-ELF object files generated by the offline compilation of OpenCL kernel files using clcc. The output of clld can be re-linked using clld as many times as necessary, but can only be linked once to produce an ELF object or executable file using the conventional linker ld. In order to protect against multiple links using ld, hash values are included in CL-ELF object files such that multiple values can only be resolved with clld. The following is a synopsis of the usage for clld.

```
clld [options] file1.o file2.o ... [-o output.o]
```

Options:

-b Include only binaries in the CL-ELF object, i.e., do not embed the original source code. This prevents the kernels from being JIT compiled on target platforms and devices not included in the offline compilation.

-h–**help** Print a brief help message.

-mall Include binaries for all devices supported by all available platforms.

-mavail Include binaries for only those devices available on the host system.

-mdevice= Select exclusive list of devices to include where is a comma separated list with no spaces. Device names are vendor specific. Note that the naming convention will in some cases employ device aliases, e.g., all x86_64 processors are identified with that simple tag regardless of the exact processor name.

-mdevice-exclude= Select list of devices to exclude where is a comma separated list with no spaces. Device names are vendor specific. Note that the naming convention will in some cases employ device aliases, e.g., all x86_64 processors are identified with that simple tag regardless of the exact processor name.

-mplatform= Select exclusive list of platforms to include where is a comma separated list with no spaces.

-mplatform-exclude= Select list of platforms to exclude where is a comma separated list with no spaces.-o

Specifiy the output filename for the final ELF object file. The default naming convention for compiling a single OpenCL kernel file is the filename base with the .o extension. The default naming convention for compiling multiple OpenCL kernel files is out_clcc.o .

-s Include only source the original source code in the CL-ELF object, i.e., do not embed any device specific binaries. This will require the kernels to be JIT compiled on all target platforms and devices.

-v Generate verbose diagnostic information.

–**version** Print version information.

6.4 clnm: Show the Contents of CL-ELF Sections

clnm is a tool analogous to the conventional nm program and allows the programer to examine the contents of the CL-ELF sections in an ELF object or executable. The following example shows the output from clnm used to examine an executable linked with OpenCL kernels compiled using clcc:

```
]clnm bdt_nbody.x
clnm: 'nbody_kern.cl' bin [amdapp:Cypress]
clnm: 'nbody_kern.cl' bin [amdapp:Cayman]
clnm: 'nbody_kern.cl' bin [amdapp:ATI RV770]
clnm: 'nbody_kern.cl' bin [amdapp:ATI RV710]
clnm: 'nbody_kern.cl' bin [amdapp:ATI RV730]
clnm: 'nbody_kern.cl' bin [amdapp:Juniper]
clnm: 'nbody_kern.cl' bin [amdapp:Redwood]
clnm: 'nbody_kern.cl' bin [amdapp:Cedar]
clnm: 'nbody_kern.cl' bin [amdapp:WinterPark]
clnm: 'nbody_kern.cl' bin [amdapp:BeaverCreek]
clnm: 'nbody_kern.cl' bin [amdapp:Loveland]
clnm: 'nbody_kern.cl' bin [amdapp:Barts]
clnm: 'nbody_kern.cl' bin [amdapp:Turks]
clnm: 'nbody_kern.cl' bin [amdapp:Caicos]
clnm: 'nbody_kern.cl' bin [amdapp:x86_64]
clnm: 'nbody_kern.cl' bin [coprthr:x86_64]
clnm: 'nbody_kern.cl' ksym copy_kern
clnm: 'nbody_kern.cl' ksym nbody_kern
clnm: 'nbody_kern.cl' ksym copy_kern
clnm: 'nbody_kern.cl' ksym nbody_kern
clnm: 'nbody_kern.cl' src [<generic>]
```

6.5 cldebug: Compute Layer Debug Interface

cldebug provides an interface to debug information generated by the various libraries and tools provided in the SDK.

```
cldebug [-v level] [-t tempdir] -- ./program [options ...]
```

-v level : set the level of reporting

-t tempdir : set the full path to a temp directory to use for JIT compilation; specifying this option will also prevent the temporary files from being removed in order to allow them to be examined

7 CLRPC: OpenCL Remote Procedure Calls

7.1 Overview of OpenCL Remote Procedure Calls (RPC)

CLRPC provides an OpenCL Remote Procedure Call (RPC) implementation that allows OpenCL host calls to be executed on remote platforms. By design this may require no changes at all to the host application when using the OpenCL loader provided with the COPRTHR SDK. For more complex and specialized applications, client host code can be linked directly with libclrpc and a few extensions are provided for specifying the remote CLRPC server or servers tht the application should be connected to.

At the present time many of the complex OpenCL semantics have been tested, e.g., `clEnqueueMapBuffer()`, that prove challenging to execute using RPC. However, the user is cautioned that not all such complex scenarios have been fully tested. For the most typical scenarios the behavior using CLRPC should be consistent with that of a local platform implementation.

An obvious question with CLRPC is whether the compute performance of an accelerator can be accessed over an ordinary network without being undermined by bandwidth issues. There is no magic to be found and, much like the early issues with the PCIe bus transfers dominating any theoretical performance gains from a GPU, the network latency and bandwidth must be addressed in order to successfully utilize CLRPC. Fortunately many examples of use cases can be identified where the use of remote compute devices can be made ot perform with an overal advantage in terms of reduced time-to-solution for the compute task at hand.

7.2 Setting up a CLRPC Server: `clrpcd`

The COPRTHR SDK provides a basic OpenCL server `clrpcd` that may be used for exporting the OpenCL support available on a platform over a network. The server will use the libocl loader to to access the available OpenCL libraries on the platform. The usage of `clrpcd` as as follows:

```
clrpcd [-a address] [-p port]
```

Options:

-a address : Specify address the server should listen on.

-p : Specify the port the server should listen on.

By default the server will bind to `127.0.0.1` port `2112`. These defaults may be overiden with the command line options as shown.
The following example shows how to run the server such that it will listen on a platforms external IP address to export the OpenCL platforms over a network,

```
clrpcd -a 192.168.1.5
```

Note: in order to properly function it is necessary to ensure that the chosen port is open. The server will export the available OpenCL platforms using RPC. At present support may be expected for most of the OpenCL 1.1 standard.

When setting up a CLRPC server it can be useful to enable verbose debugging output to monitor the operation of `clrpcd`. This can be done by simply running the server through the debug interface provided with the COPRTHR SDK,

```
cldebug -- clrpcd -a 192.168.1.5
```

7.3 Accessing Remote CLRPC Servers From OpenCL

The easiest and most powerful way in which to access OpenCL platforms that have been exported over a network is to use the features of the OpenCL loader (`libocl`) provided with the COPRTHR SDK. Specifically, one or more CLRPC servers may be specified in an `ocl.conf` file used by `libocl` to setup the platforms presented to an OpenCL appliation. The following is an example of a `clrpc` section that could be added to an ocl.conf file in order to connect to the CLPRC server setup as described above,

```
clrpc = {
    enable = "yes";
    servers = {
        { url="192.168.1.5:2112" }
    }
};
```

The *clrpc* section of an `ocl.conf` file can define multiple server connections. For more details on the `libocl` loader and `ocl.conf` files, see the section on Precise Platform Configuration: `ocl.conf` files. This raises an important issue that must be resolved in the host application when multiple server connections are defined: *exactly how should the application select the correct platform to use?* This issue goes deeper than CLRPC and exposes a problem with the OpenCL approach to the concept of a "platform" that may be decribed as the vendor platform barrier. A more complete discussion of this issue is provided below along with a solution through the use of the STDCL context `stdnpu` that includes all networked devices. The simple answer to question stated above is that one must still select a single platform, whether local or remote, based on the name of the platform, and construct an OpenCL context and get device IDs as one is normally required to do in OpenCL. If one wants to use multiple CLRPC servers this would then mean managing multiple platforms at the application level. As an alternative, the introduction of a more precise mechanism for specifying OpenCL platforms (`ocl.conf`) to be presented to an allication using the `libocl` loader can be used for controlling individual CLRPC server that an application uses. Fortunately STDCL provides a better way provided (`stdnpu`).

7.4 OpenCL Extensions for More Control

For some specialized applications it may be desireable to link to the `libclrpc` OpenCL implementation directly and set the remote CLRPC server connections from within a client application, by passing entirely the use of an OpenCL platform loader. For this purpose, extensions are provided for defining CLRPC server connections prior to the OpenCL call `clGetPlatformIDs()`.

The following example shows the current API extension. However **please note** that these extensions are not fully developed and should be expected to be changed in subsequented releases. The are described here only because they may be of interest for early experimentation by developers. Programmers are strongly encouraged to use the method described above involving the use of `ocf.conf` files for defining connections to CLRPC servers.

```
#include "CL/cl.h"
#include "clrpcd.h"

int main()
{

    clrpc_server_info servers[] = {
        { "192.168.1.5", 2112 },
        { "192.168.1.6", 2112 },
        { "192.168.1.7", 2114 },
    };

    if ( clrpc_connect(3,servers) )
        fprintf(stderr,"clrpc_connect() returned an error\n");

    ...

    clGetPlatformIDs( ... );
```

```
}
```

Note that the CLRPC servers *must* be defined prior to the `clGetPlatformIDs()` call. (This requirement should make sense upon consideration of the purpose of the `clGetPlatformIDs` call.)

7.5 A STDCL Context for All Networked Devices: `stdnpu`

One problem with CLRPC that stems from the design of OpenCL itself is that each CLRPC server will export one or more OpenCL "platforms" which are in all practical terms not interoperable in any way. This is the platform wall, an unfortunate design choice that was unncessary.

The STDCL context stdnpu resloves this issue by creating a super-context that will consist of all networked devices regardless of platform. This makes the use of networked devices in code already designed for multiple devices as simple as replacing, e.g., stdgpu with stdnpu.