Brown Deer Technology

STDCL API Reference v1.6

1 Synopsis

```
    #include <stdcl.h>

    Link with -lstdcl.

Default Contexts
    stddev, stdcpu, stdgpu, stdacc, stdrpu, stdnpu,
    clgetndev()

Dynamic CL Program loader
    clopen(), clsopen(), clsym(), clclose(), clbuild()

Memory Management
    clmalloc(), clmrealloc(), clfree(), clglmalloc(),
    clmctl(), clmctl_va(),
    clmattach(), clmdetach(), clsizeofmem(), clmsync(), clglmsync()

Kernel Management
    clndrange_init1d(), clndrange_init2d(), clndrange_init3d(),
    clarg_set(), clarg_set_local(), clarg_set_global(),
    clfork(), clforka()

Synchronization
    clflush(), clwait()

Environment Variables
    STDDEV, STDCPU, STDGPU, STDRPU,
    STD[DEV|CPU|GPU|RPU]_PLATFORM_NAME,
    STD[DEV|CPU|GPU|RPU]_MAX_NDEV,
    STD[DEV|CPU|GPU|RPU]_LOCK
```

1.1 ————-

2 Description

The STandarD Compte Layer (STDCL) is a powerful API for targeting compute offload accelerators such as GPUs and Intel MIC processors. The STDCL implementation leverages OpenCL for portability while providing simpler and more intuitive semantics. STDCL greatly reduces the amount of code required for a particular application. 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 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.

2.1 ————-

3 Application Programming Interface (API)

The STDCL interface provides support for: * Default contexts * Dynamic loader for compute-offload programs * Distributed memory management * Compute offload kernel execution * Synchronization * Extensions for CL/GL interoperability

3.1 Default Contexts

STDCL provides several default contexts similar to the default I/O streams provided by stdio. These default contexts are defined to include the most typical use-cases. Each default context is of type CLCONTEXT*. The following default contexts are provided:

```
CLCONTEXT* stddev; /* All supported devices */

CLCONTEXT* stdcpu; /* All supported multi-core CPU processors */

CLCONTEXT* stdgpu; /* All supported GPUs */

CLCONTEXT* stdacc; /* All supported accelerators (that are not CPUs or GPUs) */

CLCONTEXT* stdrpu; /* All supported reconfigurable processors */

CLCONTEXT* stdnpu; /*  Super-context containing all supported networked devices */
```

`cl_uint clgetndev( CLCONTEXT* cp )`

This call returns the number of devices in the context `cp`.

`void stdcl_init()`

This call initializes the STDCL contexts and should be made once prior to using the default contexts. This call is required for Windows only. Linux and FreeBSD will simply ignore the call.

3.2 Dynamic Loader for Compute-Offload Programs

STDCL provides a convenient interface for dynamically loading CL programs and accessing OpenCL kernels. The functions clopen(), clsym() and clclose() are designed to mirror the semantics of the more familiar functions dlopen(), dlsym() and dlclose() used to access the Linux dynamic loader. The following functions are provided for dynamically loading CL programs and accessing OpenCL kernels:

`void* clopen( CLCONTEXT* cp, const char* filename, int flags)`

This call opens a file containing the source or binary program defining one or more kernels and performs the steps necessary to compile and link the program object. A handle is returned that can be used in subsequent calls to access the actual kernels in the program. The handle is only valid within the specified context `cp`.

The flags argument allows control over the behavior of the function. The flag CLLD_NOW instructs the call to perform all of the steps involved with creating and building the program; the flag CLLD_LAZY instructs the call to defer these steps until the handle is first used. The call accepts a flag set to 0 in which case the default behavior (CLLD_NOW) is used.

If the flags argument CLLD_NOBUILD is used the compilation and build process is deferred, and a subsequent call to clbuild() must be used for the returned handle to reflect a valid (compiled and linked) program. This flag is useful when the user needs to pass in special compiler options.

By default the following compiler options will always be included:

```
-D __STDCL__
-D {__CPU__ | __GPU__}
-D {__AMD__ | __NVIDIA__ | __coprthr__}
-I $(CWD)
```

The clopen() call is only necessary for just-in-time (JIT) compilation or cross-compilation scenarios where a pre-compiled compute-offload program cannot be linked into the executable or library. For the conventional compilation model, e.g., a GCC workflow, program and kernel management is completely eliminated using the strong kernel binding available with the clcc compiler tool.

`void* clsopen( CLCONTEXT* cp, const char* srcstr, int flags )`

This call behaves exactly like clopen() with the exception that instead of providing the name of a file containing the OpenCL kernel source, the kernel code may be provided directly as a string. This call can be useful within schemes where custom kernel source is generated at run-time.

`cl_kernel clsym( CLCONTEXT* cp, void* handle, const char* symbol, int flags)`

This call takes a handle returned from a call to clopen() and returns the kernel specified by symbol. A null handle will cause the dynamic loader to search for the kernel within the CL-ELF sections of the executable itself.

The argument flags allows control over the behavior of the function. The flag `CLLD_NOW` instructs the call to perform all of the steps involved with creating the kernel; the flag `CLLD_LAZY` instructs the call to defer these steps until the kernel is first used. The call accepts a flag set to 0 in which case the default behavior (`CLLD_NOW`) is used.

As with the clopen() call, clsym() is only necessary for just-in-time (JIT) compilation or cross-compilation scenarios where a pre-compiled compute-offload program cannot be linked into the executable or library. For the conventional compilation model, e.g., a GCC workflow, program and kernel management is completely eliminated using the strong kernel binding available with the clcc compiler tool.

int clclose(CLCONTEXT* cp, void* handle)

This call decrements the reference count on the associated handle. If the reference count drops to zero then the associated program source or binary is unloaded and the associated file is closed. Under normal usage this call is used to safely release the programs created by a call to `clopen()`.

void* clbuild(CLCONTEXT* cp, void* handle, char* options, int flags)

This call is used following a call to clopen() or clsopen() with the CLLD_NOBUILD flag. Calling clbuild() will complete the process of compiling and building the kernel program. This call accepts user-specified compiler options. The handle passed in must be a valid handle created by a call to clopen() or clsopen() with the CLLD_NOBUILD flag. Calling clbuild() will complete the porocess of compilinng and building the kernel program. This call accepts user-specified compiler options. The handle passed in must be a valid handle created by a call to clopen() or clsopen() with the CLLD_NOBUILD flag. In addition to the user defined compiler options, the standard compiler options described for clopen() are also included.

3.3 Distributed Memory Management

STDCL provides functions for allocating and managing distributed memory that may be shared between the host and co-processor devices. Memory allocated with clmalloc() may be used on the host like an ordinary memory allocation obtained with a conventional malloc() call.

At the same time the allocation may be used transparently as the global memory for kernel execution on a co-processor device. The programmer uses a single pointer representing the allocated memory which may be re-attached to various CL contexts using clmattach() and clmdetach(). Memory consistency can be maintained using the clmsync() function which synchronizes memory between the host and co-processor devices.

The following functions are provided for distributed memory management.

void* clmalloc(CLCONTEXT* cp, size_t size, int flags)

This call allocates memory suitable for sharing between co-processor devices within a context. The size of the allocation is specified in bytes. The memory is not cleared. The last argument is used to pass flags to control the behavior of function. The flag CL_MEM_DETACHED may be used to allocate memory that is not attached to a context in which case cp must be 0. If flags is 0 the default behavior is to allocate memory attached to a specified context.

void* clmrealloc(CLCONTEXT* cp, void* ptr, size_t size, int flag)

This call re-allocates memory suitable for sharing between co-processor devices within a context and may be used to change the size of an existing allocation. The ptr argument must be a valid memory allocation erturned by either clmalloc(), or a previous call to clmrealloc(). The size of the allocation is specified in bytes. The memory is not cleared. The last argument is used to pass flags to control the behavior of function. The flag CL_MEM_DETACHED may be used to allocate memory that is not attached to a context in which case cp must be 0. If flags is 0 the default behavior is to allocate memory attached to a specified CL context.

`void clfree( void* ptr)`

This call frees memory allocated with clmalloc(). The memory specified by ptr can be either attached or detached from a context. Calling clfree() with ptr equal to 0 is considered an error.

`size_t clsizeofmem(void* ptr)`

This call returns the size in bytes of the memory allocated with clmalloc().

`int clmctl( void* ptr, int op, ... )`

`int clmctl_va( void* ptr, int op, va_list )`

These calls provide generalized control over a device-sharable memory allocation and differ only in the way optional arguments are passed in. the ptr argument is a pointer to device-sharable memory returned by any of the calls clmalloc(), clmrealloc(), or clglmalloc(). The following operations are presently supported.

`CL_MCTL_SET_IMAGE2D`: mark the allocation to be of OpenCL image2d_t type. The arguments are `size_t sz0, size_t sz1, size_t sz2, cl_image_format* fmt`, where `sz0` is the image width, `sz1` is the image height, `sz2` is not used an dshould be set to zero, and fmt is a pointer to an image format struct that may be set to 0 if not used.

`CL_MCTL_SET_USRFLAGS`: set the cl_mem_flags bits for the allocation based on the user mem flags value passed in. This operation is only valid on a detached memory allocation. The argument is `cl_mem_flags usrflags` corresponding to the user mem flags to be set.

`CL_MCTL_CLR_USRFLAGS`: clear the cl_mem_flags bits for the allocation based on the user mem flags value passed in. This operation is only valid on a detached memory allocation. The argument is `cl_mem_flags usrflags` corresponding to the user mem flags to be cleared.

`cl_event clmsync( CLCONTEXT* cp, unsigned int devnum, void* ptr, int flags)`

This call is used to synchronize memory between the host platform and co-processor devices. The memory specified by ptr must have been allocated by clmalloc() or an equivalent call and associated with a context.

The behavior of clmsync() is controlled by the flags argument which must be set with either CL_MEM_HOST or CL_MEM_DEVICE. These flags are mutually exclusive and it is an error to set both or none. In addition the flags CL_EVENT_WAIT and CL_EVENT_NOWAIT control the blocking behavior for the call. For a blocking call the flag CL_EVENT_NORELEASE may be specified to prevent the call from releasing OpenCL events created as a result of the call.If the flag CL_EVENT_NORELEASE is specified, the programmer is responsible for releasing the returned event with the OpenCL call clReleaseEvent().

The following examples demonstrate typical uses of clmsync():

- Non-blocking sync to device memory:

  ```
  clmsync(stdgpu,0,ptr,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
  ```

- Non-blocking sync to host memory:

  ```
  clmsync(stdgpu,0,ptr,CL_MEM_HOST|CL_EVENT_NOWAIT);
  ```

- Blocking sync to device memory:

  ```
  clmsync(stdgpu,0,ptr,CL_MEM_DEVICE|CL_EVENT_WAIT);
  ```

- Blocking sync to host with release of event:
  `clmsync(stdgpu,0,ptr,CL_MEM_HOST|CL_EVENT_WAIT);`

`cl_event clmcopy( CLCONTEXT* cp, unsigned int devnum, void* src, void* dst, int flags)`

This call is used to copy memory on an co-processor device. The memory specified by src and dst must have been allocated by clmalloc() or an equivalent call and associated with a context.

The behavior of clmcopy() is controlled by the flags argument. The flags CL_EVENT_WAIT and CL_EVENT_NOWAIT control the blocking behavior for the call. For a blocking call the flag CL_EVENT_NORELEASE may be specified to prevent the call from releasing OpenCL events created as a result of the call.If the flag CL_EVENT_NORELEASE is specified, the programmer is responsible for releasing the returned event with the OpenCL call clReleaseEvent().

`int clmattach( CLCONTEXT* cp, void* ptr )`

This call is used to attach memory allocated by clmalloc() to a context. In order to change the attachment of memory from one context to another, the memory must first be unattached using a call to clmdetach(). It is an error to call with a ptr to memory that is already attached to a context.

`int clmdetach( void* ptr )`

This call is used to detach memory from a context. The memory must have been allocated by clmalloc().

3.4 Compute Offload Kernel Execution

STDCL provides simplified interfaces for setting up the index-space and arguments for kernel execution. Executing a kernel on an co-processor device is supported using clfork() which allows blocking and non-blocking execution behavior. The following functions are provided for kernel management.

`clndrange_t clndrange_init1d( gtoff0,gtsz0,ltsz0)`

`clndrange_t clndrange_init2d( gtoff0,gtsz0,ltsz0, gtoff1,gtsz1,ltsz1)`

`clndrange_t clndrange_init3d( gtoff0,gtsz0,ltsz0, gtoff1,gtsz1,ltsz1, gtoff2,gtsz2,ltsz2)`

The clndrange_init*() functions are used to initialize a variable of type clndrange_t used to store the index-space over which a kernel is to execute. These functions will be implemented as macros to allow for struct initialization in C. The arguments gtoff, gtsz and ltsz represent the global offset, global size and local size of the index-space for a given dimension, respectively. As an example, the following initializes a two dimensional NDRange with no offsets over a global index space of size 512 by 2048 with a local work group size of 4 by 16:

```
clndrange_t ndr = clndrange_init2d( 0,512,4 0,2048,16);
```

`void clarg_set( CLCONTEXT* cp, cl_kernel krn, unsigned int argnum, Tn arg )`

This call is used to set the argument of an kernel for arguments of intrinsic non-pointer type that are to be passed by value. The size of the argument is inferred from the type of the argument and may be a vector type, e.g., cl_float4.

Explicityl setting kernel arguments is no longer required when using the clforka() variant of the clfork() call. See below for details.

`void clarg_set_global( CLCONTEXT* cp, cl_kernel krn, unsigned int argnum, void* ptr )`

This call is used to set the argument of an kernel for arguments that are pointers to global memory as defined in the OpenCL specification. The memory must have been allocated by clmalloc() in the appropriate context of the kernel.

Explicityl setting kernel arguments is no longer required when using the clforka() variant of the clfork() call. See below for details.

`void clarg_set_local( CLCONTEXT* cp, cl_kernel krn, unsigned int argnum, size_t sizeb )`

This call is used to set the argument of an kernel for arguments that are pointers to local memory as defined in the OpenCL specification. Local memory of size sizeb bytes will be allocated for use by the kernel.

Setting local memory size from the host must be done explicitly. In such a case the clforka() variant of the clfork() call cannot presently be used.

`cl_event clfork( CLCONTEXT* cp, unsigned int devnum, cl_kernel krn, clndrange_t* ndr, int flags )`

`cl_event clforka( CLCONTEXT* cp, unsigned int devnum, cl_kernel krn, clndrange_t* ndr, int flags [, arg0, ..., argN ])`

These calls are used to execute a kernel on the OpenCL device specified by devnum. In the case of clfork() the kernel arguments must be set prior to the call using the clarg_set*() functions described above. In the case of clforka() the kernel arguments may simply be added, in correct order, to the calls argument list. The kernel is executed over an index-space of work-items defined by ndr.

The behavior of clfork() may be controlled using the flags CL_EVENT_WAIT or CL_EVENT_NOWAIT. Specifying the flag CL_EVENT_NOWAIT will cause clfork() to return immediately. Specifying the flag CL_EVENT_WAIT will cause clfork() to block until the kernel execution is complete. Including the flag CL_EVENT_NORELEASE will prevent the event associated with the kernel execution to be released for blocking calls to clfork(). If the flag CL_EVENT_NORELEASE is specified the programmer is responsible for releasing the returned event with the OpenCL call clReleaseEvent().

The following examples demonstrate typical uses of clfork() and clforka():

- Blocking execution of a kernel on device number 0 automatically releasing the associated event:

  ```
  clfork( stdgpu, 0, my_krn, &ndr, CL_EVENT_WAIT);
  ```

- Non-blocking execution of a kernel on device number 2 automatically releasing the associated event:

  ```
  clfork( stdgpu, 2, my_krn, &ndr, CL_EVENT_NOWAIT);
  ```

- Blocking execution of a kernel on device number 0, setting three kernel arguments a, b, and c, and automatically releasing the associated event:

  ```
  clforka( stdgpu, 0, my_krn, &ndr, CL_EVENT_WAIT, a, b, c );
  ```

- Non-blocking execution of a kernel on device number 2, setting three kernel arguments a, b, and c, and automatically releasing the associated event:

  ```
  clforka( stdgpu, 2, my_krn, &ndr, CL_EVENT_NOWAIT, a, b, c);
  ```

3.5 Synchronization

STDCL provides functions for synchronization to manage the inherently asynchronous operations associated with the use of co-processors.

`int clflush( CLCONTEXT* cp, unsigned int devnum, int flags )`

This call is used to flush all commands enqueued in the command queue associated with the co-processor device specified by the device number devnum within the specified context. For typical OpenCL implementations leverged by STDCL this can be necessary to force the execution of commands without blocking on the host. A call to clflush() is non-blocking and will return immediately. At present the argument flags should be set to 0.

`cl_event clwait( CLCONTEXT* cp, unsigned int devnum, int flags )`

This call is used to block on the completion of all commands enqueued in the command queue associated with the co-processor device specified by the device number devnum within the specified context.

The flags argument is used to control the behavior of the call as follows. The flag CL_KERNEL_EVENT will cause the call to block on completion of all enqueued kernel events enqueued by calls to clfork(). the flag CL_MEM_EVENT will cause the call to block on completion of all enqueued memory events enqueued by call to clmsync(). The flags CL_KERNEL_EVENT and CL_MEM_EVENT may be combined in a single call. Including the flag CL_EVENT_NORELEASE will prevent all OpenCL events to be released before clwait() returns. If the flag CL_EVENT_NORELEASE is specified the programmer is responsible for releasing all events with the OpenCL call clReleaseEvent().

The following examples demonstrate typical uses of clwait():

- Block on completion of all kernel execution events on device number 0 releasing all events:

  ```
  clwait( stdgpu, 0, CL_KERNEL_EVENT );
  ```

- Block on completion of all memory events on device number 2 releasing all events:

  ```
  clwait( stdgpu, 2, CL_MEM_EVENT );
  ```

- Block on completion of all kernel and memory events on device number 2 releasing all events:

  ```
  clwait( stdgpu, 2, CL_ALL_EVENT );
  ```

3.6 Extensions for CL/GL interoperability

`void* clglmalloc( CLCONTEXT* cp, cl_GLuint glbufobj, cl_GLenum target, cl_GLint miplevel, int flags)`

This call allocates memory suitable for sharing between co-processor devices within a context based on an existing OpenGL memory object glbufobj. The OpenGL memory object can be either a memory buffer, a 2D texture, a 3D texture or a render buffer. The additional argunts target and miplevel are used for textures and ignored oterwise. The size of the allocation is implied by the OpenGL memory size. The memory is not cleared. The last argument flags is used to select the type of OpenGL memory object and control the behavior of function. This argument must include one and only one of the following: CL_MEM_GLBUF, CL_MEM_GLTEX2D, CL_MEM_GLTEX3D, CL_MEM_GLRBUF. The flag CL_MEM_DETACHED may be used to allocate memory that is not attached to a context in which case cp must be 0. If flags is 0 the default behavior is to allocate memory attached to a specified context.

`cl_event clglmsync( CLCONTEXT* cp, unsigned int devnum, void* ptr, int flags)`

This call is used to sync memory between device-sharable memory and OpenGL buffers.

The flags argument must be set to either CL_MEM_CLBUF or CL_MEM_GLBUF to define the destination of the sync operation.

3.7 Environment Variables

The run-time behavior of STDCL can be controlled using environment variables as follows.

`STDDEV, STDCPU, STDGPU, STDRPU`

Each default context is can be controlled by the associated environment variable. A value of 0 or 1 will disable or enable the context, respectively. The default behavior is to attempt to enable the context if valid devices are available within the selected platform.

`STD[DEV|CPU|GPU|RPU]_PLATFORM_NAME`

Set the platform name for the desired platform to be used for a given context. If none is provided or the specified platform is unavailable, a default will be selected.

`STD[DEV|CPU|GPU|RPU]_MAX_NDEV`

Set the maximum number of devices for a given context regardless of whether more devices exist for the selected platform. If there is an insufficient number of devices, the maximum available will be provided.

`STD[DEV|CPU|GPU|RPU]_LOCK`

Set a lock ID for the process to enforce exclusive access to the provided devices across all processes run with the same lock ID. This feature is primarily useful to ensure the multiple MPI processes on a multi-GPU platform are each given exclusive access to a GPU with no requirement on the application itself.

`COPRTHR_OPENCL_PROFILING_ENABLE`

Setting this environment variable will cause STDCL contexts using OpenCL to enable OpenCL event profiling.

3.8 ————-

4 Examples

4.1 Example #1

The following example shows the use of STDCL for a simple program that adds two vectors on a GPU or a CPU:

```
/* example #1 */

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

#define SIZE 1024

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

   int i;

   CLCONTEXT* cp = (stdgpu)? stdgpu : stdcpu;

   void* clh = clopen(cp, "add_vec.cl",CLLD_NOW);
   cl_kernel k_addvec = clsym(cp, clh, "addvec_kern", CLLD_NOW);

   float* aa = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* bb = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* cc = (float*)clmalloc(cp,SIZE*sizeof(float),0);

   for(i=0;i<SIZE;i++) {
      aa[i] = 111.0f * i;
      bb[i] = 222.0f * i;
   }

   bzero(cc,SIZE*sizeof(float));

   clndrange_t ndr = clndrange_init1d(0,SIZE,64);

   clmsync(cp,0,aa,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,0,bb,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
```

```
   clforka(cp,0,k_addvec,&ndr,CL_EVENT_NOWAIT, aa, bb, cc);

   clmsync(cp,0,cc,CL_MEM_HOST|CL_EVENT_NOWAIT);

   clwait(cp,0,CL_MEM_EVENT|CL_KERNEL_EVENT);

   for(i=0;i<SIZE;i++) printf("%f %f %f\n",aa[i],bb[i],cc[i]);

   if (aa) clfree(aa);
   if (bb) clfree(bb);
   if (cc) clfree(cc);

   clclose(cp,clh);
}
```

4.2 ————-

4.3 Example #2

The following example shows the use of STDCL for a simple program that adds two vectors on two GPUs:

```
/* example #2 */

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

#define SIZE 1024

int main()
{
   stdcl_init(); // required for Windows only, Linux and FreeBSD will ignore this call

   int i,n;

   CLCONTEXT* cp = stdgpu;

   void* clh = clopen(cp, "add_vec.cl",CLLD_NOW);
   cl_kernel k_addvec = clsym(cp, clh, "addvec_kern", CLLD_NOW);

   float* aa[2];
   float* bb[2];
   float* cc[2];

   aa[0] = (float*)clmalloc(cp,SIZE*sizeof(float)/2,0);
   aa[1] = (float*)clmalloc(cp,SIZE*sizeof(float)/2,0);
   bb[0] = (float*)clmalloc(cp,SIZE*sizeof(float)/2,0);
   bb[1] = (float*)clmalloc(cp,SIZE*sizeof(float)/2,0);
   cc[0] = (float*)clmalloc(cp,SIZE*sizeof(float)/2,0);
   cc[1] = (float*)clmalloc(cp,SIZE*sizeof(float)/2,0);
```

```
   for(i=0;i<SIZE/2;i++) {
      aa[0][i] = 111.0f * i;
      aa[1][i] = 111.0f * (SIZE/2 + i);
      bb[0][i] = 222.0f * i;
      bb[1][i] = 222.0f * (SIZE/2 + i);
   }

   bzero(cc[0],SIZE*sizeof(float));
   bzero(cc[1],SIZE*sizeof(float));

   clndrange_t ndr = clndrange_init1d(0,SIZE/2,64);

   clmsync(cp,0,aa[0],CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,1,aa[1],CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,0,bb[0],CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,1,bb[1],CL_MEM_DEVICE|CL_EVENT_NOWAIT);

   clforka(cp,0,k_addvec,&ndr,CL_EVENT_NOWAIT,aa[0],bb[0],cc[0]);

   clmsync(cp,0,cc[0],CL_MEM_HOST|CL_EVENT_NOWAIT);

   clflush(cp,0,0);

   clforka(cp,1,k_addvec,&ndr,CL_EVENT_NOWAIT,aa[1],bb[1],cc[1]);

   clmsync(cp,1,cc[1],CL_MEM_HOST|CL_EVENT_NOWAIT);

   clflush(cp,1,0);

   clwait(cp,0,CL_MEM_EVENT|CL_KERNEL_EVENT);
   clwait(cp,1,CL_MEM_EVENT|CL_KERNEL_EVENT);

   for(i=0;i<SIZE/2;i++) printf("%f %f %f\n",aa[0][i],bb[0][i],cc[0][i]);
   for(i=0;i<SIZE/2;i++) printf("%f %f %f\n",aa[1][i],bb[1][i],cc[1][i]);

   if (aa[0]) clfree(aa[0]);
   if (aa[1]) clfree(aa[1]);
   if (bb[0]) clfree(bb[0]);
   if (bb[1]) clfree(bb[1]);
   if (cc[0]) clfree(cc[0]);
   if (cc[1]) clfree(cc[1]);

   clclose(cp,clh);
}
```
