OpenCL Emu Documentation

Table of Contents

1. Introduction

OpenCL Emu is a set of effective tools for the OpenCL software development designed for AMD GPUs without the explicit need of GPU hardware. It allows developing and debugging an OpenCL kernel as a C++ procedure inside your MS Visual Studio application while providing an easy switch between CPU, GPU or GPU-emulator at the backend.

This tool would enable programmers to start developing using OpenCL language instantly without having to learn the intricacies of the OpenCL run-time, saving them time to concentrate more on developing parallel algorithms and making the shift to OpenCL a step easier.

2. Installing & Configuring OpenCL Emu

2.1 Prerequisites

Before proceeding to the installation, please ensure the following requirements.

i. AMD Accelerated Parallel Processing (APP) SDK.
ii. AMD OpenCL Emulator Installation Files.
iii. AMD GPU with OpenCL support recommended but not required.
iv. Microsoft Visual Studio 2008.
v. Microsoft Windows XP or later.

2.2 Download & Install OpenCL Emu

i. Install the AMD APP SDK to your computer.
ii. Download the OpenCL Emu and preferably place the opencl_emu" and "opencl_emu_app" folders inside the Stream SDK folder and name it "OpenCL Emulator".

2.3 Setting up Environment Variables

As part of the installation, the environment variable CLEMU_ROOT needs to be setup manually to point to the installation directory. This can be done easily using the SET command from command prompt. e.g.

```
SET CLEMU_ROOT=C: \Program Files\ATI Stream\opencl_emu
```

2.4 Setting up Microsoft Visual Studio Test Project

The Microsoft Visual Studio is the recommended developing environment for this tool and requires these quick changes to configure the OpenCL Emulator.

2.4.1 Additional Include Directories:

From the Visual Studio window, go to: Project Properties -> C/C++ -> General and include the following new paths in the additional include directories:

```
$(ATISTREAMSDKROOT)\include
$(CLEMU_ROOT)\clemu
$(CLEMU_ROOT)\runCL
$(CLEMU_ROOT)\SDKUtil
$(CLEMU_ROOT)\include
```

2.4.2 Preprocessor:

Now go to "Proprocessor" tab in the same Project Properties window and add the following preprocessor

```
_GPU_EMU_IMPL
```

2.4.3 Code Generation:

The code generation option would be selected depending upon the target application. For example if you're building a Multi-threaded Debug application (not DLL) then you may select:

```
Multi-Threaded Debug (/MTd).
```

2.4.4 Linker:

Now proceed to ***Configuration Properties*** -> ***Linker*** -> ***General*** and add the following as additional library directories:

```
$(ATISTREAMSDKROOT)\lib\x86
$(CLEMU_ROOT)\lib\debug\x86
$(CLEMU_ROOT)\lib\x86
```

Proceed ***to Linker*** -> ***Input*** and add the following as Additional Dependencies:

```
OpenCL.lib
runCl.lib
```

Your Visual Studio is now all set to run OpenCL Emu.

To build libraries and sample applications, use a solution file located under:

…\opencl-emu_app\src

It does not require any changes and should correctly compile out of the box.

3. How to Use OpenCL Emu with Example

3.1 OpenCL-emu Macro Definitions

3.1.1 Kernel declaration/definition

These are some of the kernel Macro definitions which have been used inside this tool. This can be used for reference purposes as well.

kernel NAME	__Kernel(NAME)
kernel NAME with statically defined 1D group size	__KernelGrpSz1(NAME, GROUP_SZ1)
kernel NAME with statically defined 2D group size	__KernelGrpSz2(NAME, GROUP_SZ1, GROUP_SZ2)

3.1.2 Kernel arguments declaration/definition

An argument of type TYPE and of name NAME	__Arg(TYPE, NAME)
First argument in an argument list	__ArgFirst(TYPE, NAME)
List argument in an argument list	__ArgLast(TYPE, NAME)
kernel has no arguments	__ArgNULL
statically delcared local array	__Local(TYPE, NAME, SIZE)
Last statement of the kernel – mandatory	__Return

An example of a kernel without arguments:

```
__Kernel(hello)
__ArgNULL
{
...
__Return
}
```

The static attribute of an internal kernel procedure is recommended to avoid name conflicts inside C++ environment. You may use other ways to avoid it as well. For example, to use prefixes specific to the kernel's .cpp file:

```
__STATIC
```

OpenCl long/ulong types

```
__LONG

__ULONG
```

3.2 callCL Application:

3.2.1 Features

callCL is the basic procedure that is essentially an easy way of calling OpenCL kernels. This method combines the earlier procedure that included creating a context, making queues, allocating buffers and finally calling a kernel into a single step. This in turn decreases the overhead and complexity of calling OpenCL kernels. The function supports cpu, gpu and gpu_emu calls.

3.2.2 Parameter Definitions

The callCL function has the following definition:

```
int callCL(const char * _device_type,
               int _domainDim,
               int _domain[],
               int _group[],
               const char * _program_location,
               const char * _program,
               const char*_kernel_entry_name,
               ClKrnlArg* _args = 0);
```

`_device_type`	Target device. Possible values: "cpu", "gpu" , "gpu_emu"
`_domainDim`	Work Dimension
`_domain[]`	Nd Range, Global worksize
`int _group[]`	Local workgroup size
`_program_location`	Kernel directory *(example: "c:/opencl_emu/test/")*
`_program`	Filename containing kernels *(example: "testKernel.cl")*
`_kernel_entry_name`	Specific kernel name
`_args`	Arguments passed to the kernel

callCl also provides 3 distinctive defines to help isolating pieces of code written for a specific platform:

- `_GPU_EMU_IMPL`
- `CPU_IMPL`
- `GPU_IMPL`

To separate a Cedar class GPU from its higher-end siblings the runCL provides another definition:

- `CEDAR`

3.2.3 Developing Kernels in callCL

The primary feature of callCL is to make it easier to create kernels while keeping them fully compatible with the OpenCL standards. When using with physical devices, i.e. CPU or GPU, OCL_EMU does everything for you on the OpenCL application side. It initializes OpenCL, allocates memory buffers and runs a kernel. In order to use buffers with callCL, you need allocate the needed buffers in the system memory. In emulation mode - callCl("gpu_emu",…) - _global buffer pointers you see inside the kernel will be the same system pointers you get when you allocate the buffers.

Furthermore, OCL_EMU writes data into an OpenCL buffer from your system buffer, if needed. (The POPULATE flag needs to be added to the INPUT or INPUTOUTPUT buffer for such memory transfers). The pointer you would send to callCL will be system memory pointer. If the POPULATE flag has been added to an INPUTOUTPUT or OUTPUT buffer the OCL_EMU reads data back into your system buffer for further use and the data could be passed to the next kernel(s).

OCL_EMU also provides the capability to mix "cpu", "gpu", "gpu_emu" kernels in the same pipeline and in any order you require, making a very fast prototype of a complicate pipeline much easier. In general, the benefit of OCL_EMU would be to help develop, correct and optimize kernels as well as to verify them in a complicate pipeline. After a kernel has been perfected, the developer may plug it into the OpenCL application, ready to run on commodity hardware.

One use case scenario would be to start with writing the kernel code inside the emulator. The best situation is when you can design and code the kernel skeleton with (almost) all kernel parameters known at the time. At this moment you may plug the kernel into your OCL_EMU, setup a breakpoint inside the kernel and call callCL("gpu_emu",…). The goal is to run the kernel as a C++ procedure inside the OCL_EMU environment and to verify your preliminary assumptions: input/output data formats, data structures, kernel parameters' number and values. At this stage you may also try to run callCl(…) with a "cpu" or/and "gpu" parameter to convince yourself that your kernel is properly compiled in OpenCL. After that it's better to return back to "gpu_emu" for developing and testing your kernel until you feel it's correct. Since the OpenCl programming language is a restricted version of C99, I'd also advise to switch to "cpu"/"gpu" parameters periodically to make sure your code is syntactically correct from the OpenCl language point of view.

Note: If you do not have a full OpenCL application in place yet, you may use *AMD Stream KernelAnalyzer* to verify an exact OpenCL language compliance.

However KernelAnalyzer does not except –I parameter defining addition include directories, at least for now.

To overcome the restriction one can add this piece of code inside the kernel:

```
// When working in KA, uncomment the first line, else comment it.
//#define KA
#ifdef KA
#include
// absolute location of the clemu_opencl.h file in your system
"C:\Program Files\ATI Stream\opencl_emu\clemu\clemu_opencl.h"
#else
#include "clemu_opencl.h"
#endif
```

Notes:

If you already have the C-Model of your algorithm, you can easily compare it with the data flow in your kernel with the help of the emulator. Otherwise, since you have a C –like language with some extensions and a full blown debugger in the shape of OpenCL, you may prefer to develop a kernel inside the emulator from scratch and use **it as a C-model**. After you've decided that your emulated kernel is logically correct, you may run it on CPU or GPU without the emulator.

It's recommended to first replace "gpu_emu" with "cpu" and run. You may encounter some compilation problems since the OpenCL language is stricter than VC++ and you may miss something. After compilation problems have been solved, if any, and the kernel runs, you may compare your simulated and OpenCL results. In case it's successful, you may switch to "gpu". Do not forget, however, that you may encounter the difference between the CPU and GPU backend due to different synchronization schemes and other subtle differences between the backend compilers. If you are convinced that your first kernel is correct, you may develop a next kernel and add it to the pipeline. The first kernel can run in "cpu" or "gpu" mode but the new kernel might still be in "gpu_emu" mode. Flag POPULATE has been designed to keep memory consistency, it adds a copy time though. If your pipeline is correct, you may move it into your real OpenCL app.

3.2.4 Moving Kernels From callCL

There are some steps involved before you can move your kernels written in OpenCL Emu to your real application. To avoid changing your kernel, e.g. OCL_EMU macros, you need to do ONE of the three procedures defined below:

1. copy file "clemu_opencl.h" from <*OPENCL_EMU_DIRECTORY>\opencl_emu\clemu"* into the directory where you are keeping your kernel.

2. Add the following path:

   –I …<OPENCL_EMU_DIRECTORY>\opencl_emu\clemu to the OpenCL compiler option string.

3. Use CLEMU_ROOT environment variable to do the same as in point 2, but programmatically. Here is how you can do just that:

```
char  *env_root;
size_t var_len;
char root_location[_MAX_PROGRAM_NAME + 1];
char Options[1024] = {0};

// read environment varaible
    _dupenv_s(&env_root,&var_len, "CLEMU_ROOT");

      if ( !var_len )
      {
        return(SDK_FAILURE);
      }

      strcpy_s(root_location,var_len,env_root);

    if ( root_location[var_len-1] != '\\' && root_location[var_len-1] != '/')
      {
         root_location[var_len-1] = '\\';
         root_location[var_len] = 0;
      }
      strcat_s(root_location, _MAX_PROGRAM_NAME, "clemu");
      sprintf_s(Options,512," -I \"%s\"", root_location);
```

3.2.5 Switching between CEDAR-type & High-end GPUs

You may also choose to add platform specific defines to your application. This would allow an easy switch between CEDAR type and high-end AMD GPUs.

That's how callCL does it (plus CEDAR define), DeviceName has been found using OpenCL INFO APIs, DevType has been sent as a parameter:

```
if (!strcmp(GetDevType(),"cpu"))
{
      strcat_s(Options, 256, " -DCPU_IMPL=1 ");
}
else if (!strcmp(GetDevType(),"gpu"))
{
      strcat_s(Options,256,  " -DGPU_IMPL=1 ");
      if (!strcmp(DeviceName, "Cedar"))
             {
              strcat_s(Options,256,  " -DCEDAR");
             }
}
```

To emulate AMD GPU devices more precisely the OML_EMU has a property files. The current implementation requires by-hand switching to the Cedar-type GPU emulation from its high-end siblings.

To do so the programmer has to change the following line in the file

…\ opencl_emu\clemu\ clemu_intnl.hpp from

```
#define DEFAULT_DEVNAME "juniper"
```
to
```
#define DEFAULT_DEVNAME "cedar"
```

3.2.6 Sample Applications with Use Cases

OpenCL Emu comes with 5 sample applications located in:

…\opencl_emu_app\src\cl\app:

- NBody
- NBodyEmu
- SimpleImage
- SimpleImageEmu

…\opencl_emu_app\src\cp_cl\app:

- HelloCl

Out of these, the following are built as GPU Emulation (Debug):

- NBodyEmu
- SimpleImageEmu
- HelloCl

The target device type can be specified in the *–device* command argument:

```
-device {gpu, cpu, gpu_emu}
```

Notes:

- If GPU Emulation mode **(gpu_emu)** is specified as the target device type then you can put breakpoints in the kernel itself which will break the execution at that point during a debugging session. For example, any breakpoint/s set inside the HelloCl_Kernels.cpp source file will always stop the execution at the point during a debugging session.
- All kernel source files have the naming convention of **[NAME]_Kernel.cpp**

Sample Application: NBody

```
#include <CL/cl.h>
#include "runCL.h"

/* An absolute path or a path relative to the default or explicitly defined working
directory (see Visual Studio property page\Debugging) might be NULL. */

#define DEFAULT_KERNEL_LOCATION ".. \\src"

int NBody::runCLKernels()
{

cl_int status;
ClKrnlArg Args[1024];

/* Set appropriate arguments to the kernel */


/* Create memory objects for position */
_ArgFirst(Args, CL_ARG_INPUTOUTPUT_PTR | CL_ARG_POPULATE_PTR,cl_float*,pos,
               numBodies*sizeof(cl_float4));

/* Create memory objects for velocity */
_Arg(Args,CL_ARG_INPUTOUTPUT_PTR | CL_ARG_POPULATE_PTR,cl_float*, vel,
       numBodies*sizeof(cl_float4));

/* numBodies */
_Arg(Args,CL_ARG_VALUE,cl_int,numBodies, sizeof(cl_int));

/* time step */
_Arg(Args,CL_ARG_VALUE,cl_float,delT, sizeof(cl_float));

/* upward Pseudoprobability */
_ArgLast(Args,CL_ARG_VALUE,cl_float,espSqr, sizeof(cl_float));

/* local memory */
//_ArgLast(Args,CL_ARG_LCL_MEM_SZ,int,GROUP_SIZE * 4 * sizeof(float),sizeof(int));


int globalThreads[] = {numBodies};
int localThreads[] = {groupSize};


status = callCL(deviceType.c_str(), 1, globalThreads, localThreads, DE-
FAULT_KERNEL_LOCATION, "NBodyEmu_Kernels.cpp", "nbody_sim",  Args);
//     status = callCL("gpu_emu", 1, globalThreads, localThreads, "FaceRe-
cogn_Kernels.cpp", "nbody_sim",  Args);

return SDK_SUCCESS;
}
```

NBody Kernel Source:

```
#include "clemu_opencl.h"

/*
__kernel void nbody_sim(__global float4* pos,
                                    __global float4* vel,
                                    int numBodies,
                                    float deltaTime,
                                    float epsSqr,
                                    __local float4* localPos);
*/

// dynemically defined local memory as argument: __ArgLast(__local float4*,
localPos)
__Kernel(nbody_sim)
         __ArgFirst(__global float4*, pos)
         __Arg(__global float4*, vel)
             __Arg(uint, numBodies)
             __Arg(float, deltaTime)
             __ArgLast(float, epsSqr)
{
    unsigned int tid = get_local_id(0);
    unsigned int gid = get_global_id(0);
    unsigned int localSize = get_local_size(0);

    // Number of tiles we need to iterate

    unsigned int numTiles = numBodies / localSize;

      // statically declared local memory
    __Local(float4, localPos, 256);

    // position of this work-item

    float4 myPos = pos[gid];
    float4 acc = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

      //Continued...
   for(int i = 0; i < (int)numTiles; ++i)
    {

        // load one tile into local memory
            //int idx = i * localSize + tid;
        uint idx = mad24( (uint)i, localSize, tid);
        localPos[tid] = pos[idx];

        // Synchronize to make sure data is available for processing
        barrier(CLK_LOCAL_MEM_FENCE);

        // calculate acceleration effect due to each body
        // a[i->j] = m[j] * r[i->j] / (r^2 + epsSqr)^(3/2)
        for(int j = 0; j < (int)localSize; ++j)
        {
            // Calculate acceleartion caused by particle j on particle i
            float4 r = localPos[j] - myPos;
            float distSqr = r.x * r.x  +  r.y * r.y  +  r.z * r.z;
            float invDist = 1.0f / sqrt(distSqr + epsSqr);
            float invDistCube = invDist * invDist * invDist;
            float s = localPos[j].w * invDistCube;

            // accumulate effect of all particles
            acc += ((float4)s * r);
        }

        // Synchronize so that next tile can be loaded
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    float4 oldVel = vel[gid];

    // updated position and velocity
    float4 newPos = myPos + oldVel * deltaTime + acc * 0.5f * deltaTime *
deltaTime;
    newPos.w = myPos.w;

    float4 newVel = oldVel + acc * deltaTime;

    // write to global memory
    pos[gid] = newPos;
    vel[gid] = newVel;

      // MANDATORY
      __Return;
}
```

4. Limitations

4.1 Not Supported Modes

This release of **OpenCL Emu** has the following programming limitations:

- Modes: You cannot create 3D domains. Only 1D and 2D domains are supported.
- Kernel functions cannot be called within another kernel.
- Image Type: Filtering is not supported
- Expressions of the type: **.s6789i.e. only .s0, .s1, .s3, .even, .odd**
- Vector Initialization: Instead of initializing vectors like this:
  ```
  uint2 val2 = uint2(v0, v1);
  ```

  Do it like this:
  ```
  uint2 val2;
        val2.x = v0;
        val2.y = v1;
  ```

  And when you want to initialize a vector with the same value then instead of doing it like this:
  ```
  uint2 val00 = uint2(val0, val0);
  ```

  Do it like this:
  ```
  uint2 val00 = (uint2)val0;
  ```

Both these methods are valid but the later ones are preferred because of their OCL_EMU compiler friendliness. This same pattern applies to vectors of all dimensions e.g. *uint3*, *float2* etc.

4.2 Known Issues

- OpenCL Emulator cannot use the same system memory buffer pointers as different callCL arguments.
- Both static and dynamic local memory declarations cannot be used in the same callCL call, use either one of these.

5. Future Releases