# OpenCL Emu Documentation

Table of Contents

[OpenCL Emu Documentation 1](#_Toc287875324)

[1. Introduction 2](#_Toc287875325)

[2. Installing & Configuring OpenCL Emu 2](#_Toc287875326)

[2.1 Prerequisites 2](#_Toc287875327)

[2.2 Download & Install OpenCL Emu 2](#_Toc287875328)

[2.3 Setting up Environment Variables 2](#_Toc287875329)

[2.4 Setting up Microsoft Visual Studio Test Project 3](#_Toc287875330)

[3. How to Use OpenCL Emu with Example 4](#_Toc287875331)

[3.1 OpenCL-emu Macro Definitions 4](#_Toc287875332)

[3.2 callCL Application Interface: 5](#_Toc287875333)

[3.2.1 Features 5](#_Toc287875334)

[3.2.2 Parameter Definitions 6](#_Toc287875335)

[3.2.3 Developing Kernels in callCL 6](#_Toc287875336)

[3.2.4 Moving Kernels From callCL 9](#_Toc287875337)

[3.2.5 Switching between CEDAR-type & High-end GPUs 10](#_Toc287875338)

[3.2.6 Sample Applications with Use Cases 11](#_Toc287875339)

[4. Notes and Limitations 15](#_Toc287875340)

[4.1 Notes 15](#_Toc287875342)

[4.2 Limitations 15](#_Toc287875343)

[4.3 Known Issues 16](#_Toc287875344)

# 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.

# Installing & Configuring OpenCL Emu

## Prerequisites

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

1. AMD Accelerated Parallel Processing (APP) SDK.
2. AMD OpenCL Emulator Installation Files.
3. AMD GPU with OpenCL support recommended but not required.
4. Microsoft Visual Studio 2008.
5. Microsoft Windows XP or later.

## Download & Install OpenCL Emu

1. Install the AMD APP SDK to your computer.
2. Download the OpenCL Emu into any location (*<OPENCL\_EMU\_DIRECTORY>)* on your local machine.

## 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. In order to add the new Environment variable, navigate to:  
*Control Panel\System and Maintenance\System\Advanced System Setting\Environment Variables*

And add:

Variable: CLEMU\_ROOT  
Value: *<OPENCL\_EMU\_DIRECTORY>*\opencl\_emu

## 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.

#### 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:

$(AMDAPPSDKROOT)\include  
$(CLEMU\_ROOT)\clemu  
$(CLEMU\_ROOT)\runCL  
$(CLEMU\_ROOT)\SDKUtil  
$(CLEMU\_ROOT)\include

Note:

Useres who had downloaded the ocl-emu before APP SDK 2.4 have to add environment variable

AMDAPPSDKROOT = %ATISTREAMSDKROOT%

#### Preprocessor:

Now go to “Preprocessor” tab in the same Project Properties window and add the following preprocessor

\_GPU\_EMU\_IMPL

#### 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).

#### Linker:

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

$(AMDAPPSDKROOT)\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.

# How to Use OpenCL Emu with Example

This section explains, with an example, how to use OCL\_EMU tool to quickly right OpenCL kernels.

## OpenCL-Emu Macro Definitions

OpenCL-Emu offers the following set of macros that an application developer can use:

#### Kernel declaration/definition

|  |  |
| --- | --- |
| **Macro Name** | **Macro Definition** |
| \_\_Kernel(NAME) | kernel name specified as NAME parameter |
| \_\_KernelGrpSz1(NAME, GROUP\_SZ1) | kernel named ‘NAME’ with statically defined 1D group size |
| \_\_KernelGrpSz2(NAME, GROUP\_SZ1, GROUP\_SZ2) | kernel named ‘NAME’ with statically defined 2D group size |

#### Kernel arguments declaration/definition

|  |  |
| --- | --- |
| \_\_Arg(TYPE, NAME) | An argument of type TYPE and of name NAME |
| \_\_ArgFirst(TYPE, NAME) | First argument in an argument list |
| \_\_ArgLast(TYPE, NAME) | List argument in an argument list |
| \_\_ArgNULL | Kernel has no arguments |
| \_\_Local(TYPE, NAME, SIZE) | Statically declared local array |
| \_\_Return | Last statement of the kernel – mandatory |

An example of a kernel without arguments:

Example of a kernel with arguments

\_\_Kernel(nbody\_sim)

\_\_ArgFirst(\_\_global float4\*, pos)

\_\_Arg(\_\_global float4\*, vel)

Arg(uint, numBodies)

\_Arg(float, deltaTime)

\_ArgLast(float, epsSqr)

{

\_\_Return;

}

Example of a kernel with a statically declared group size:

\_\_KernelGrpSz1(amdPh1,64)

\_\_ArgFirst(\_\_global const inp\*, \_cachedCascade)

\_\_Arg(uint, \_Width)

\_\_Arg(uint, \_Height)

\_\_Arg(\_\_global uint \*, \_out)

\_ArgLast(\_\_local uint \*,\_localMemPtr)

{

\_\_Return;

}

To avoid name conflicts between internal procedures in C++ environment it’s preferable to use **inline** attribute or OCL-EMU**\_\_STATIC** macro. For example:

\_\_STATIC   
void myInternalProcedure(…){  
…  
}

OR

inline  
void myInternalProcedure(…){  
…  
}

OpenCl long/ulong types: To be able to use OpenCL long/ulong types and their vectored derivatives use OCL\_EMU macro replacement:

\_\_LONG / \_\_ULONG

## callCL Application Interface:

### Features

callCL is the only interface that your application will require to invoke and make use of OCL\_EMU. The method combines an initializing OpenCl device(or OCL\_EMU emulator), creating an OpenCl context, making queues, allocating buffers and finally calling a kernel into a single step. This has been done to decrease complexity of calling OpenCL kernels. The function supports cpu, gpu and gpu\_emu devices.

### Parameter Definitions

The callCL function has the following definition:



|  |  |
| --- | --- |
| \_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.cpp”)* |
| \_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

### 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 them 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, it’s advisable 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.For KernelAnalyzer to find clemu\_opencl.h the following compile option has to be added:

–I *<OPENCL\_EMU\_DIRECTORY>\clemu*

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.

### 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.

1. Use CLEMU\_ROOT environment variable to do the same as in point 2, but programmatically. Here is how you can do just that:



### 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:



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

to  


### **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:



#### 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

##### NBody Kernel Source:

#### 

# Notes and Limitations



## Notes

1. Since VC++ does not have OpenCl-like memory attributes: **\_\_global, \_\_local, \_\_constant,** the C++ compiler cannot catch their omissions, be aware.
2. Always insert barrier( ) function after using \_\_local memory – read or write.

Also insert barrier( ) after using atomic. Be sure that all working\_items in the group hit barrier() the same number of times as required by OpenCl specification.

1. When using global scope constant, please add \_\_STATIC attribute, for example:

\_\_STATIC \_\_constant myConstArray[..] = {…, …}

## Limitations

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

* Only 1D and 2D domains are supported.
* Kernel functions cannot be called within another kernel.
* Image Type: Filtering is not supported
* Vector Initialization: Instead of initializing vectors like this:  
  

Do it like this:  


And when you want to initialize a vector with the same value then instead of doing it like this:  


Do it like this:  


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.

## 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.