**OpenCL –emu is a set of effective tools for the OpenCl kernel development.**

It allows to develop and to debug a kernel a C++ procedure inside your MS Visual Studio and easily switch between cpu, gpu or gpu-emulator backend.

The emulator intends to help at least 2 kinds of developers.

First group are novices who do not like to learn intricacies of the OpenCl run-time and would like to get “Hello World” kernel as an one line call (as it should be). The second group is algorithm developers who also do not like to know idiosyncrasies of OpenCl run-time but need to concentrate on intricacies of their massive parallel algorithm(s).

**REBUILDING OpenCL –emu on your local machine and attaching it to your application.**

Setup and build OpenC

Prerequisite:

AMD OpenCl SDK has to be installed.

…\opencl\_emu

…\opencl\_emu\_app

Have to be downloaded and located on the same hierarchical level and under the same parent directory.

Environment variable CLEMU\_ROOT has to point to the location of the ...\opencl\_emu directory.

For example:

SET CLEMU\_ROOT = C:\...\depot\main\sw\advtech\tools\opencl\_emu

**The OpenCl EMU VS9 test solution setup.**

The solution is located under

…\opencl-emu\_app\src

It does not require any changes besides “Code generation” property (not always , see below) and should be correctly compiled out-of-the-box.

Code generation

run-time libraries - depends on the target application

example: Multi-threaded Debug (/MTd)

**YOUR Application’s VS9 solution setup.**

Visual Studio property page.

C/C++

Additional include directories

"$(ATISTREAMSDKROOT)\include";"$(CLEMU\_ROOT)\clemu";"$(CLEMU\_ROOT)\runCL; "$(CLEMU\_ROOT)\SDKUtil";"$(CLEMU\_ROOT)\include"

Code generation

run-time libraries - depends on the target application

example: Multi-threaded Debug (/MTd)

Preprocessor

\_GPU\_EMU\_IMPL;

Linker

Additional library directories

$(ATISTREAMSDKROOT)\lib\x86; $(CLEMU\_ROOT)\lib\debug\x86;$(CLEMU\_ROOT)\lib\x86

Additional libraries

OpenCL.lib runCl.lib

**OpenCl-emu macros used in kernels.**

**Kernel declaration/definition**

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)

**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)

Lirst 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

Example of 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 way to avoid it. For example, to use prefixes specific to the kernel’s .cpp file

\_\_STATIC

OpenCl long/ulong types

\_\_LONG

\_\_ULONG

**Application - callCL.**

callCl is a simplfied way of calling OpenCl (gpu, cpu or gpu\_emu) kernels. It instantiates OpenCl, creates a context, queues, allocates buffers and launches a kernel.

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 - "cpu", "gpu" or "gpu\_emu"

// OpenCl domain definition

int \_domainDim

int \_domain[]

int \_group[]

// kernel directory

\_program\_location

// kernel source file name, i.e. NBodyEmu\_Kernels.cpp

\_program

// kernel entry point

\_kernel\_entry\_name

// array of arguments

\_args

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 define.

CEDAR

**callCl .h file**

#include "runCL.h"

**Tips and clarifications.**

To use buffers with callCl you just allocate them in the system memory. The pointer you are sending to callCl is a system memory pointer. When calling callCl(“gpu\_emu”,…) , \_global buffer pointers you see inside the kernel are the same system pointers you’ve got when you allocate the buffers.

When calling callCl(“cpu” or “gpu”,…) , the OpenCl-emu does everything for you on the OpenCl application side - it initializes OpenCl, allocates buffers and runs a kernel. It writes data into an OpebCl buffer from your system buffer, if needed. This to happen the flag POPULATE has to be added to the INPUT or INPUTOUTPUT buffer. If the flag has been added to an INPUTOUTPUT or OUTPUT buffer the OpenCl-emu reads data back into the system buffer for further use and the data could be passed to the next kernel(s). It give you ability to mix “cpu”, “gpu”, “gpu\_emu” kernels in the same pipeline any way you want, making a very fast prototype of a very complicate pipeline much easier.

In general, the openCl-emu’s purpose at the current stage is to help to develop correct and optimized kernels and verify them in a complicate pipeline. After it’s been done a developer may plug the kernel into his/her full (real) OpenCl app.

**Author’s tips how to do (just) that.**

I usually 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 OpenCl-emu pipeline, setup a breakpoint inside the kernel and call callCL(“gpu\_emu”,…).

The goal is to run the kernel as a C++ procedure inside the OpenCl-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 prperly 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.

The best situation is if you have a c-model already. With the help of the emulation you may easily compare the data flow in your c-simulator and inside the kernel.

However since what you have in your possession is a C –like language with some extensions and a full blown debugger 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 “real” HW.

I’d recommend, first, replace “gpu\_emu” with “cpu” and run.

You may encounter some compilation problems since the OpenCl language is more strict than VC++ and you may miss something.

After compilation problem has been solved the kernel runs and you may compare your simulated and OpenCl results

If it’s successful you may switch to “gpu”.

Do not forget, however, you may encounter the difference between the cpu and gpu backend due to different synchronization scheme and other subtle differences between backend compilers.

If you are convinced 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 still might be in “gpu\_emu” mode. Flag POPULATE has been designed to keep a memory consistency. It adds a copy time though.

If your pipeline is correct you may move it into your real OpenCL app.

When you are moving your kernels into your real app to avoid changing any code inside your kernel (i.e., OpenCl-emu macros), you may do one of 3 things:

1. copy file

clemu\_opencl.h

from

…\ depot\main\sw\advtech\tools\opencl\_emu\clemu

into the directory where you are keeping your kernels.

2. add

–I …<YOUR\_ABSOLUTE\_LOCATION>\clemu line into the OpenCl compiler option string.

3. or use CLEMU\_ROOT environment variable to do the same as in 2., but programmatically.

Here is an approximate code to 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);

You may also choose to add platform specific defines to your application. 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 more precisely emulate AMD GPU devices the OpenCl\_emu has a property files. Unfortunately the current implementation requires by-hand switching to the Cedar-type GPU emulation from its highe-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"

Note.

The current implementation supports a proper compilation and a proper emulation run of kernel .cpp files only if they are compiled and linked on the application level. It does not preclude kernel .cpp files to be located anywhere in the development tree. In other words currently OpenCl-emu does not allow kernel .cpp files to be linked inside static libraries or dlls. The OpenCl-emu kernels as C++ procedures have to run in the same thread the main application does.

Sample app

#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 "C:\\AlexProjects\\rsh\\src\\... [\\src](file://\\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, DEFAULT\_KERNEL\_LOCATION, "NBodyEmu\_Kernels.cpp", "nbody\_sim", Args);

// status = callCL("gpu\_emu", 1, globalThreads, localThreads, "FaceRecogn\_Kernels.cpp", "nbody\_sim", Args);

return SDK\_SUCCESS;

}

sample kernel

#include "clemu\_opencl.h"

/\*

\_\_kernel

void

nbody\_sim(

\_\_global float4\* pos ,

\_\_global float4\* vel,

int numBodies,

float deltaTime,

float epsSqr,

\_\_local float4\* localPos)

\*/

\_\_Kernel(nbody\_sim)

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

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

\_\_Arg(uint, numBodies)

\_\_Arg(float, deltaTime)

\_\_ArgLast(float, epsSqr)

// dynemically defined local memory as argument

// \_\_ArgLast(\_\_local float4\*, localPos)

{

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);

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;

}

Not supported.

3D domain (only 1D and 2D).

kernel calling kernel

image type currectly supports only RGBA, CL\_SIGNED\_INT32, CL\_UNSIGNED\_INT32 and CL\_FLOAT

any expressions of type:

.s6789, i.e. only .s0, .s1, .s3

.even

.odd

known problems

do not use the same system memory buffer pointers as different callCL arguments.

use either static or dynamic local memory declaration not both in the same callCl.