A report of playing with OpenCL Conformance Candidate:

findings, questions and suggestions

by Oscar B. G. on 21 May 2009.

Hi I have been playing for a week with the OpenCL conformance candidate (on Windows and Visual Studio 2008) and here are my findings, questions and suggestions about OpenCL Nvidia implementation and comparisons to CUDA..
Sorry for the long report..

Part 1: About the JIT compiler

First thing to say : A feature I'm very proud of OpenCL is the builtin compiler vs the CUDA compilation steps.. I know that since CUDA 2.1 CUDA ships with a JIT compiler but this only compiles PTX code.. I know that this JIT theoretically can provide the same benefits of a higher level language JIT..
but this is a feature future versions of CUDA can provide.. i.e. extending the JIT API/features to be able to compile kernels in C for CUDA (not a general .cu file since it can contain intermixed CPU/GPU code) would be nice since it is also the model of compilation used by shader languages: Cg and GLSL for example..

I have modified the example oclTranspose for obtaining intermediate binaries of the kernels transpose and tranpose_naive
doing :
clGetProgramInfo (cpProgram,CL_PROGRAM_BINARIES,0,ptx_code,&ret);
I didn't know what would obtain if a PTX file or a CUBIN, in fact I obtained a PTX file.
The first lines are:

```
// Generated by NVIDIA PTX Backend for LLVM
.version 1.5
.target sm_13, texmode_independent

/* Global Launch Offsets */
.const[0] .s32 %global_block_offset[3];
..
/* Temporary variables for v2load/v4load/read */
.local .b8 vector_load_8[4];
..
/* Function Prototypes */

/* extern unsized array args for kernel 'transpose' */
.extern .shared .b8 transpose_param_4[]; /* ptxbe_def_block */
.const[0] .u32 transpose_param_4_offset;


.const[0] .b32 %dummy_const; /* needed to avoid an assert in driver */

/* Function Bodies */
.entry transpose (
    .param .b32 transpose_param_0,
    .param .b32 transpose_param_1,
    .param .u32 transpose_param_2,
    .param .u32 transpose_param_3)
.maxntid 16, 16, 1
{
    .reg .b32 ptxbe_def_odata;
    .reg .b32 ptxbe_def_idata;
```

This lines deserve some comments of the code:

1. The PTX version is 1.5, which is undocumented currently (the latest is 1.4

and ships with CUDA 2.2). Extrapolating
from recent releases that new CUDA version implies PTX version +=0.1; then this
PTX version would be used in CUDA 2.3 compiler.
But the PTX code seems to have new nomenclatures and also different size memory
spaces. I claim that becuase I compiled the transpose sample
from CUDA sample.
That includes the same kernels but expressed in each language i.e.:
for example this line in OpenCL:
 xIndex = get_group_id(1) * BLOCK_DIM + get_local_id(0);
is:
 xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
which ideally should map to the same PTX instructions.
Some PTX code of the C for CUDA transpose kernels:

```
.version 1.4
        .target sm_13
        // compiled with C:\CUDA\bin/../open64/lib//be.exe
        // nvopencc built on 2009-05-02

        .reg .u32 %ra<17>;
        ..
        .entry _Z9transposePfS_ii (
                .param .u64 __cudaparm__Z9transposePfS_ii_odata,
                .param .u64 __cudaparm__Z9transposePfS_ii_idata,
                .param .s32 __cudaparm__Z9transposePfS_ii_width,
                .param .s32 __cudaparm__Z9transposePfS_ii_height)
        {
```

comparing the two PTX generated code I have found for example that the name of
the function is well preserved in OpenCL:
.entry transpose
vs. in CUDA PTX:
.entry _Z9transposePfS_ii

Also since the signature of the kernel is:
void transpose(float *odata, float *idata, int width, int height)

seems that device pointers (the first two parameters) correspond in CUDA to :
.param .u64
(suggesting 64bit spaces?)
vs.
.param .b32
in OpenCL.
(suggesting 32bit spaces?)

My primary concern about this differences in PTX (and confirmated in practice
see below) is that
two kernels (device functions) with the same name and signature should map to
similar PTX code indistinctly of
the high level language and at least should be usable the compiled PTX code from
the other high level language.
This feature is expected in CPU programs where a function written in C or C++ or
Fortran once compiled
say to assembly language/object file should be callable from other languages
provided the right name and signature of the function
(and also preventing name mangling issues).

For testing this concern about PTX interoperabilty I modified OpenCL code
for building the kernels form the binaries instead of the .cl file:
clCreateProgramWithBinary (cxMainContext,1,&device,tams,(const unsigned char
**)ptx_code,bin, &ciErrNum);

Using the PTX generated earlier from
clGetProgramInfo (cpProgram,CL_PROGRAM_BINARIES,0,ptx_code,&ret);
results in correct functioning of the program.
To test about PTX interoperabilty I used PTX generated from C for CUDA transpose
kernels.

This to my surprise this resulted running correctly(?) much further than expected but still no OK. clCreateProgramWithBinary and clBuildProgram functioning correctly.
Also surprinsingly (?) clCreateKernel(cpProgram, "transpose", NULL); runs correctly since the name of the C for CUDA kernel in the PTX is _Z9transposePfS_ii.
Then I found that clCreateKernel(cpProgram, "transpose2", NULL); still runs correctly but this funciton doesn't exist so clCreateKernel seems to not return errors on inexistant fuctions (at least on PTX of CUDA code).
The first sign of an error is that
clGetKernelInfo (ckKernel,CL_KERNEL_NUM_ARGS,0,&args,NULL);
returns 0 arguments so something is failing.
Ignoring it results in executing the kernel (wtih clEnqueueNDRangeKernel) returning a CL_OUT_OF_RESOURCES.

More interestingly indicating the naming of the kernel in C for CUDA PTX code results in an error earlier specifically at:
clCreateKernel(cpProgram, "_Z9transposePfS_ii", &errcode_ret);
 returning the CL_INVALID_KERNEL_DEFINITION error.

The improved interop between CUDA and OpenCL generated PTX code would be useful and at least provide two benefits:

1. Reuse of existing PTX code from CUDA applications or libraries shipping with no source code or avoiding conversion of source codes of the kernels (altough that seems to be pretty simple). Also this would allow to improved or earlier delivered features in this case of NVIDIA GPUs
present in nvcc and more slowly coming to the standard or as Nvidia/Khronos OpenCL extensions. For example this would allow now to use compiled C for CUDA kernels wich use doubles (sm_13) and currently not exposed in OpenCL (at least the extension it isn't present).
2. PTX translators to other ISA (x86/SSE/AVX/LNI or Cell or AMD Stream IL) some which are currently in development by researchers and would not need to track diffrences about PTX for CUDA or PTX of OpenCL. After all PTX is meant to be a virtual instruction set independent of the high level languages and compilers which generated it.

I have seen some other bugs in the compilation phase (adding to these PTX limitations expressed up to now):

1. A created executable when running inside Visual Studio 2008 halts at clBuildProgram and not continues (with F5 start debgguing).
Altought is running correctly running inside of Visual Studio 2008 (Ctrl+F5 no debugging) or from expolorer.
 I don't see this behavour if program is created from clCreateProgramWithBinary instead of clCreateProgramSource.

2. I suspect that if in the OpenCL code are present two kernels with names such that one is prefix of another there can be problems,
specifically a I ported transposenew kernel from the CUDA 2.2 new sample and I get a error (at clCreateKernel of one of the kernes) and then renaming to tranposnew miracously fixed that.
Specifically I had three kernels in a cl file : transpose, transpose_naive and transposenew.
Doing:
clCreateKernel(cpProgram, "transpose_navie", NULL);
clCreateKernel(cpProgram, "transpose", NULL);
clCreateKernel(cpProgram, "transposenew", NULL);
results in error in clCreateKernel(cpProgram, "transpose", &err) of type CL_INVALID_KERNEL_DEFINITION
renaming transposenew to transposnew
and doing :
clCreateKernel(cpProgram, "transpose_navie", NULL);
clCreateKernel(cpProgram, "transpose", NULL);
clCreateKernel(cpProgram, "transposnew", NULL);
results in no error of the three kernels.

3. Executing a OpenCL program can fail. It fails at clBuildProgram and seems to

be related to the builtin compiler malfunctioning depending of
some enviroment variables (I believe it's PATH enviroment). Some message return
from clGetProgramBuildInfo:

```
Loading program 'reduce_kernel.cl'...
Error: Failed to build program executable!
:69: error: expected identifier or '('
\binw;C:\Program Files (x86)\ATI\ATI Brook+ 1.4.0_beta\sdk\lib;C:\M
^
:69: error: expected '=', ',', ';', 'asm', or '__attribute__' after declarator
\binw;C:\Program Files (x86)\ATI\ATI Brook+ 1.4.0_beta\sdk\lib;C:\M
      ^
:69: error: expected '=', ',', ';', 'asm', or '__attribute__' after declarator
\binw;C:\Program Files (x86)\ATI\ATI Brook+ 1.4.0_beta\sdk\lib;C:\M
                                                              ^
```

In fact the kernel file only has 68 lines and compiler error messages includes
:69: error (yeah also I'm using ATI Stream SDK and that help me find an OpenCL
SDK bug! Yeah that's fun! :-))

Now some other questions:

Part 2. About Device info and extensions:
==========

I have extended deviceQuery sample to print all the info exposed by the struct.
On a Geforce GTX 275 the info is below.
Some questions of not expected behaivor:

1. CL_DEVICE_MAX_MEM_ALLOC_SIZE returns 234881024 or aprrox 256Mbytes. This flag
points only to a single allocation i.e "malloc" ?
i.e I can obtain more than that using multiple allocations ? I suppect that yes.

2. CL_DRIVER_VERSION: 185.68, bad I have 185.85.

3. CL_DEVICE_VENDOR_ID: 4318 it's the PCI Vendor ID or what?

4. CL_DEVICE_ADDRESS_BITS: 32. What does this mean? It's that subject to change
to 64 since I find

5. IMAGE2D_MAX_WIDTH 65536   IMAGE2D_MAX_HEIGHT 32768
  I expected 8192 as OpenGL/DX10 max 2D texture dimensions. It's this high
values correct.

6. CL_DEVICE_ERROR_CORRECTION_SUPPORT 1
  What does this ECC mean?. If it's of device memory, I expected that Geforce
chips didn't come with ECC RAM and was
reserved to Quadro/Tesla.

7. The meaning of CL_EXEC_NATIVE_KERNEL of CL_DEVICE_EXECUTION_CAPABILITIES can
be misleading at first since without reading the spec I expected this to be the
capacity of executing compiled programs (in case of Nvidia PTX). But seems to be
a core capability of the spec for the FULL PROFILE and indicated by
CL_compiler_avaiable.

8. CL_DEVICE_SINGLE_FP_CONFIG
I expected stronger capabalities until I found it's related only to single
precision.
What about a device info describing double precesion capabilites if cl_khr_fp64
extension it's present since it would support more for exmaple CL_FP_FMA -
IEEE754-2008

9. Extensions.
 At least is missing
cl_khr_fp64,cl_khr_int64_base_atomics,cl_khr_int64_extended_atomics,cl_khr_byte_
addressable_store
 and cl_khr_fp16.
All these ones are supported by GT200 on CUDA 2.2.

What about cl_khr_3d_image_writes?
It's that supported on current hardware (GT200) and if yes, it's currently exposed in CUDA ?

Also I suspect that currently supported 2d image writes are equivalent to the writing texture capabilities exposed in CUDA 2.2 (texture from pitch linear memory)..

Part 3. About pinned memory
==========

I'm a bit concerned about the possibility of efficient memory transactions between host and device (aka pinned mem transfers in CUDA) or al least about understating the concept of memory object.
I have seen bandwithtest that contains the option of creating pinned memory objects(?).
and seems to equal to creating a buffer with
clCreateBuffer with CL_MEM_ALLOC_HOST_PTR instead of CL_MEM_USE_HOST_PTR.
In spec CL_MEM_ALLOC_HOST_PTR is:
This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory.

What this means?
1.Fist question: this buffers allocalted with clCreateBuffer and CL_MEM_ALLOC_HOST_PTR flag is device mem or host mem? I suppect it's always device mem since clCreateBuffer buffers's are prefixed with d_.

other option:
2. The memory is allocated on host memory and used within the device (as zero copy). I don't believe this is the case since I tested on a 8800GT and this works.

Since I believe the response is option 1 I don't understand the meaning of pinned memory on this sample since in CUDA pinned memory is special *HOST memory* allocated with special instructions and in this sample the distinction of pinned mem is a flag for allocation *DEVICE memory* .
If we see the H2D code host mem is normally allocated :
h_idata = (unsigned char *)malloc(memSize);

Also there in OpenCL seems to be the concept of mapped or direct memory. For me it seems that the more similar concept to CUDA pinned memory is OpenCL direct memory.

this results in H2D transfer:
clEnqueueWriteBuffer(cqCommandQue, d_odata, CL_FALSE, 0, memSize, h_idata, 0, NULL, NULL);

with host memory allocated wtih simple malloc.

I also see that this memory results are lower than CUDA case 5GB/s vs 2GB/s I first thinked

Resuming, my question is: Is there a way of using pinned/page locked host mem in OpenCL for efficient mem transfers?

Part 4. NVIDIA OpenCL implementation and relation to CUDA
==========

Also the other enhancements in CUDA would be exposed in OpenCL as vendor extensions or inherited transparently in NVIDIA implementation in OpenCL whre apropiate or not exposed at all?
Also seems natural to mix CUDA/OpenCL code for extracting power of some features in CUDA not presently in OpenCL that at least seem than can be interoperable at at high level of abstaction (assuming OpenCL is layered upper in the CUDA stack).

I.e. would be any option to zero copy from OpenCL?

It would be exposed as an extension or since OpenCL seems to use CUDA as backend can I create from CUDA a host mapped buffer and use the device pointer as parameter to OpenCL kernels and that kernels would use zero copy memory transparentlly.
This would allow to write portable GPGPU programs in OpenCL but use vendor specific optimizations (in this case the zero copy feature of NIVIDA latest cards).

This question is similar to the question about mixing PTX kernels indistinctively between C for CUDA and OpenCL assuming equal function names and signatures.

Also what about write cacheable or portable pinned memory?
Similar what's the situation in this case. The are plans to be exposed as extensions or by default all pinned memory would be portable between devices. Or also initializating CUDA library for use of portable pinned memory would map to portable pinned memory in OpenCL,etc..

About fences and barriers seem the relation between CUDA and OpenCL is:

1. __syncthreads() = barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);
in the general case but in some cases of the code you can simplify to:
barrier(CLK_LOCAL_MEM_FENCE); or barrier(CLK_GLOCAL_MEM_FENCE);

In this case OpenCL seems to provide more granurality for barriers.

2. mem_fence=__threadfence

In this case OpenCL offers more granularity of mem fences in two dim. in what memory hierarchy (CLK_LOCAL_MEM_FENCE
CLK_GLOBAL_MEM_FENCE) and the operations (read or write) to ensure consistency.

It's possible on Nvidia cards of using this granularity at the hardware level i.e. can we expect better perf.?

i.e. using barrier(CLK_LOCAL_MEM_FENCE) instead of
barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);

Also seems that __threadfence_block() has no similar in OpenCL and it's must be more lightweight than __syncthreads() since
__syncthreads() implies __threadfence_block().

Part 5. OpenCL and OpenGL

From what I have seen the source code is prepared for using advanced OpenCL/OpenGL interop i.e Sharing Memory Objects with OpenGL.
It's used if a GL_INTEROP is defined.
I have define and the OpenGL samples fail.
Any roadmap on when it will be working?
Also or I'm missing some part but I think WGL_create_context must be extended for support of creation of OpenGL contexts that can share
buffers with OpenCL. At least some documentation would be useful for creating a wrapper around the creation of such contexts or it's going
Nvidia to release a modified GLUT with supports creation of that contexts since GLUT is used in these samples and the creator works for NVIDIA.

Part 6. Tools

What about debbuggers and profilers?
Also what about a semi-automatic conversion of CUDA to OpenCL source codes?

Appendix A: GT200 device info

```
 Device GeForce GTX 275:
 CL_DEVICE_NAME:                                 GeForce GTX 275
 CL_DEVICE_VENDOR:                               NVIDIA Corporation
 CL_DEVICE_PROFILE:                              FULL_PROFILE
 CL_DRIVER_VERSION:                              185.68
 CL_DEVICE_VERSION:                              OpenCL 1.0
 CL_DEVICE_TYPE:                                 CL_DEVICE_TYPE_GPU
 CL_DEVICE_MAX_COMPUTE_UNITS:                    30
 CL_DEVICE_MAX_WORK_ITEM_SIZES:                  65535 / 65535 / 1
 CL_DEVICE_MAX_WORK_GROUP_SIZE:                  512
 CL_DEVICE_MAX_CLOCK_FREQUENCY:                  1403 MHz
 CL_DEVICE_IMAGE_SUPPORT:                        1
 CL_DEVICE_GLOBAL_MEM_SIZE:                      896 MByte
 CL_DEVICE_LOCAL_MEM_SIZE:                       16 KByte
 CL_DEVICE_QUEUE_PROPERTIES:
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
 CL_DEVICE_QUEUE_PROPERTIES:                     CL_QUEUE_PROFILING_ENABLE
 CL_DEVICE_VENDOR_ID:                            4318
 CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:             3
 CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR:          4
 CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT:         4
 CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT:           4
 CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG:          4
 CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT:         4
 CL_DEVICE_ADDRESS_BITS:                         32
 CL_DEVICE_MAX_MEM_ALLOC_SIZE:                   234881024
 CL_DEVICE_MAX_READ_IMAGE_ARGS:                  128
 CL_DEVICE_MAX_WRITE_IMAGE_ARGS:                 8
 CL_DEVICE_IMAGE2D_MAX_WIDTH:                    65536
 CL_DEVICE_IMAGE2D_MAX_HEIGHT:                   32768
 CL_DEVICE_IMAGE3D_MAX_WIDTH:                    2048
 CL_DEVICE_IMAGE3D_MAX_HEIGHT:                   2048
 CL_DEVICE_IMAGE3D_MAX_DEPTH:                    2048
 CL_DEVICE_MAX_SAMPLERS:                         16
 CL_DEVICE_MAX_PARAMETER_SIZE:                   4352
 CL_DEVICE_MEM_BASE_ADDR_ALIGN:                  256
 CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE:             4
 CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE:            0
 CL_DEVICE_GLOBAL_MEM_CACHE_SIZE:                0
 CL_DEVICE_GLOBAL_MEM_SIZE:                      939524096
 CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE:             65536
 CL_DEVICE_MAX_CONSTANT_ARGS:                    9
 CL_DEVICE_ERROR_CORRECTION_SUPPORT:             1
 CL_DEVICE_SINGLE_FP_CONFIG:                     CL_FP_INF_NAN
CL_FP_ROUND_TO_NEAREST
 CL_DEVICE_GLOBAL_MEM_CACHE_TYPE:                CL_NONE
 CL_DEVICE_LOCAL_MEM_TYPE:                       CL_LOCAL
 CL_DEVICE_PROFILING_TIMER_RESOLUTION:           1000ns.
 CL_DEVICE_ENDIAN_LITTLE:                        1
 CL_DEVICE_AVAILABLE:                            1
 CL_DEVICE_COMPILER_AVAILABLE:                   1
 CL_DEVICE_EXECUTION_CAPABILITIES:               CL_EXEC_KERNEL
 CL_DEVICE_EXTENSIONS:                           cl_khr_global_int32_base_atomics

cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics
cl_khr_local_int32_extended_atomics
```