MAKING OPENCL™ SIMPLE WITH HASKELL

Benedict R. Gaster
AMD
Software Architect
The ideas and work presented here are in collaboration with:
- Garrett Morris (AMD intern 2010 & PhD student Portland State)
- Garrett is also speaking at AFDS and you should check him out in session 2910
AGENDA

- Motivation
- Bringing OpenCL to Haskell
- Lifting to something more in the spirit of Haskell
- Quasi-quotation
- Questions
MOTIVATION
```c
__constant char hw[] = "Hello World\n";

__kernel void hello(__global char * out) {
    size_t tid = get_global_id(0);
    out[tid] = hw[tid];
}
```
HELLO WORLD OPENCL C SOURCE

```c
__constant char hw[] = "Hello World\n";
__kernel void hello(__global char * out) {
    size_t tid = get_global_id(0);
    out[tid] = hw[tid];
}
```

- This is a separate source file (or string)
- Cannot directly access host data
- Compiled at runtime
HELLO WORLD - HOST PROGRAM

// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
        CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
        devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
        0, NULL);

memobjs[0] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        sizeof(cl_char)*strlen("Hello World"), NULL,
        NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
        &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
        sizeof(cl_mem));

// set work-item dimensions
global_work_size[0] = strlen("Hello World");

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
        NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[0],
        CL_TRUE, 0, strlen("Hello World") * sizeof(cl_char),
        dst, 0, NULL, NULL);
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
    CL_DEVICE_TYPE_GPU, NULL, NULL, NULL, NULL);

// get the list of GPU devices associated with context
cl_devices = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
    NULL, &cb);
devices = malloc(cb);
cGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
    devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
    0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_char)*strlen("Hello
    World"), srcA, NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
    &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
    NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
    NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE,
    0, n*sizeof(cl_float), dst, 0, NULL, NULL);

Define platform and queues
Define Memory objects
Create the program
Build the program
Create and setup kernel
Execute the kernel
Read results on the host
hstr = "Hello world\n"
hlen = length hstr + 1

prog = [$cl|
    __constant char hw[] = $hstr;
    __kernel void hello(__global char * out) {
        size_t tid = get_global_id(0);
        out[tid] = hw[tid];
    }
|]

main :: IO ()
main = withNew prog $
    using (bufferWithFlags hwlen [WriteOnly]) $ \b ->
    do [k] <- theKernels
        invoke k b `overRange` ([0], [hwlen], [1])
        liftIO . putStrLn . map castCCharToChar . fst =<< readBuffer b 0

    (hlen - 1)
USING HASKELL OUR GOAL IS TO WRITE THIS

```haskell
hstr = "Hello world\n"
hlen = length hstr + 1

prog = [\$cl|
  __constant char hw[] = $hstr;
  __kernel void hello(__global char * out) {
    size_t tid = get_global_id(0);
    out[tid] = hw[tid];
  }
]main :: IO ()main = withNew prog $
  using (bufferWithFlags hlen [WriteOnly]) $ \b ->
    do [k] <- theKernels
      invoke k b `overRange` ([0], [hlen], [1])
      liftIO . putStrLn . map castCCharToChar . fst =<< readBuffer b 0 (hlen - 1)
```

- Single source
- OpenCL C code statically checked at compile time
- No OpenCL or Haskell compiler modifications
LEARN FROM COMMON USES

- In OpenCL we generally see:
  - Pick single device (often GPU or CL_DEVICE_TYPE_DEFAULT)
  - All “kernels” in cl_program object are used in application

- In Cuda the default for runtime mode is:
  - Pick single device (always GPU)
  - All “kernels” in scope are exported to the host application for specific translation unit, i.e. calling kernels is syntactic and behave similar to static linkage
BRINGING OPENCL TO HASKELL
**THE BASICS - HELLO WORLD**

```haskell
main = do p: _ <- getPlatforms
        putStrLn . (
            "Platform is by: " ++
        ) =<< getPlatformInfo p
        PlatformVendor
        c <- createContextFromType
            (pushContextProperty ContextPlatform p noProperties) (bitSet [GPU])
        p <- createProgramWithSource c . ([]) =<< readFile "hello_world.cl"
        ds <- getContextInfo c ContextDevices
        buildProgram p ds ""
        k <- createKernel p "hello"
        b :: Buffer CChar <- createBuffer c (bitSet [WriteOnly]) hwlen
        setKernelArg k 0 b
        q <- createCommandQueue c (head ds) (bitSet [])
        enqueueNDRangeKernel q k [0] [hwlen] [1] []
        putStrLn . map castCCharToChar =<<
            enqueueBlockingReadBuffer q b 0 (hwlen - 1) []
```
MAPPING OPENCL STATIC VALUES

```haskell
class Const t u | t -> u
  where value :: t -> u

data PlatformInfo t
  where PlatformProfile :: PlatformInfo String
        PlatformVersion :: PlatformInfo String
        PlatformName :: PlatformInfo String
        PlatformVendor :: PlatformInfo String
        PlatformExtensions :: PlatformInfo String

instance Const (PlatformInfo t) CLPlatformInfo
  where value PlatformProfile = 0x0900
        value PlatformVersion = 0x0901
        value PlatformName = 0x0902
        value PlatformVendor = 0x0903
        value PlatformExtensions = 0x0904
```

```haskell
data Platform_ 
  where Platform = Ptr Platform_ 

type CLPlatformInfo = Word32
```
MAPPING OPENCL API

getPlatforms :: IO [Platform]
getPlatforms = appendingLocation "getPlatforms" $
    getCountedArray clGetPlatformIDs

getPlatformInfo :: Platform -> PlatformInfo u -> IO u
getPlatformInfo platform plInf =
    case plInf of
        PlatformProfile       -> get
        PlatformVersion       -> get
        PlatformName          -> get
        PlatformVendor        -> get
        PlatformExtensions    -> get
    where get = appendingLocation "getPlatformInfo" $
        getString (clGetPlatformInfo platform (value plInf))
AND SO ON FOR THE REST OF OPENCL API

- Standard use of Haskell FFI to implement the calls in an out of Haskell into the OpenCL C world:

```haskell
foreign import stdcall "cl.h clGetPlatformIDs" clGetPlatformIDs
:: CLCountedArrayGetter(Platform)

foreign import stdcall "cl.h clGetPlatformInfo" clGetPlatformInfo
:: Platform -> CLPlatformInfo -> CLGetter
```

- Simple extensions to handle OpenGL interop, with the HOpenGL (GLUT) packages. Allows performance, directly from Haskell, close to original C version
LIFTING TO SOMETHING MORE
IN THE SPIRIT OF HASKELL
SO FAR NOTHING THAT INTERESTING

- **HOpenCL Native API**
  - Standard native function interface within the IO monad
  - Little advantage over programming in C++ or PyOpenCL

- **HOpenCL Contextual API**
  - Monad transformers to lift HOpenCL Native API beyond the IO monad
  - Quasi- quotation to lift OpenCL source into Haskell as a first class citizen
MOVE TO AN IMPLICIT MODEL

Context

CPU

GPU

Programs
Kernels
Memory Objects
Command Queues

In Order Queue
Out of Order Queue

Compute Device
• Context is an environment
• CommandQueue is shared for read/write device memory and executing kernels, easily stored implicitly with an environment
**CONTEXTUAL/QUEUED MONADS**

- Function like reader monads for OpenCL `Context` and `CommandQueue` objects, avoiding having to pass `Contexts` and/or `CommandQueues` as the first parameter to many OpenCL operations.

- Introducing new classes gets around the dependency on the `MonadReader` class that would prevent asserting both `MonadReader Context m` and `MonadReader CommandQueue m` for the same type `m`. 
EMBEDDING MONADS

- Wraps class - provides a uniform way to embed one computation into another, for example:

```haskell
with :: MonadIO m => Context -> ContextM t -> m t
```

embeds a computation in `ContextM` (a contextual monad) into any IO monad.
EMBEDDING MONADS

- In fact, the only way we expect to use a context/command queue/etc. is to populate a contextual/queued computation, so we provide a function that combines construction in the outer monad with computation in the inner monad:

\[
\text{withNew} :: \text{MonadIO } m \Rightarrow m \text{ Context} \rightarrow \text{ContextM } t \rightarrow m t
\]

- This mechanism is extensible: the quasi-quotiation support uses a single data structure to contain context, queue, and device information; however, by adding it to the 'Contextual', 'Queued', and 'Wraps' classes we can use the same code we would have with the more conventional initialization.
Many OpenCL objects are reference counted, with `clRetainXXX` and `clReleaseXXX` functions. We overload `retain` and `release` operators for all reference counted CL objects.

We can use those to build C#-like control structures to automatically release objects: `using` for newly constructed objects and retaining for parameters:

```haskell
using :: (Lifespan t, MonadIO m) => m t -> (t -> m u) -> m u
retaining :: (Lifespan t, MonadIO m) => t -> (t -> m u) -> m u
```

In fact, the `withNew` function also uses the `Lifespan` class to automatically release the objects used to run the sub-computation.
QUASI-QUOTATION
THE ROAD TO SINGLE SOURCE

- Embedded OpenCL C concrete syntax into Haskell with quasi-quotation

- Set of default quasi-quoters:
  - [$cl| ... |] OpenCL default device, command queue and so on
  - [$clCPU| ... |] OpenCL CPU device, command queue, and so on
  - [$clALL| ... |] all OpenCL devices, with N command queues, and so on
    - and on and on … (it would be nice to automatically compute these)

- Statically check OpenCL C source

- Anti-quoting can be used to reference Haskell values

- Save OpenCL binaries for reloading at runtime (guarantees correctness)
HELLO WORLD REVISITED

hstr = "Hello world\n"
hlen = length hstr + 1

prog = [\$cl|
    __constant char hw[] = $hstr;
    __kernel void hello(__global char * out) {
        size_t tid = get_global_id(0);
        out[tid] = hw[tid];
    }
|
]main :: IO ()
main = withNew prog $
    using (bufferWithFlags hwlen [WriteOnly]) $ \b ->
    do [k] <- theKernels
       invoke k b `overRange` ([0], [hlen], [1])
       liftIO . putStrLn . map castCCharToChar . fst =<< readBuffer b 0 (hlen - 1)
SIMPLIFIED VERSION

data CLMod = CLModule String DeviceType
    deriving(Show, Typeable, Data)

parseCLProg :: Monad m => Src -> String -> m CLModule

initCL :: CatchIO m => CLModule -> m Module
initCL (CLModule src dtype) =
    do (p:_) <- CL.platforms
        withNew (CL.contextFromType p [dtype]) $ 
            using (CL.programFromSource src) $ \prog ->
                do c <- CL.theContext
                    ds <- CL.queryContext ContextDevices
                    CL.buildProgram prog ds ""
                    ks <- kernels prog
                    q <- queue (head ds)
                    return (Mod c ks q)
QUESTIONS
Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREFIELD, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

© 2011 Advanced Micro Devices, Inc. All rights reserved.