Making OpenCL™ Simple with Haskell

Benedict R. Gaster
January, 2011

AMD

Attribution and WARNING

- The ideas and work presented here are in collaboration with:
  - Garrett Morris (AMD intern 2010 & PhD student Portland State)

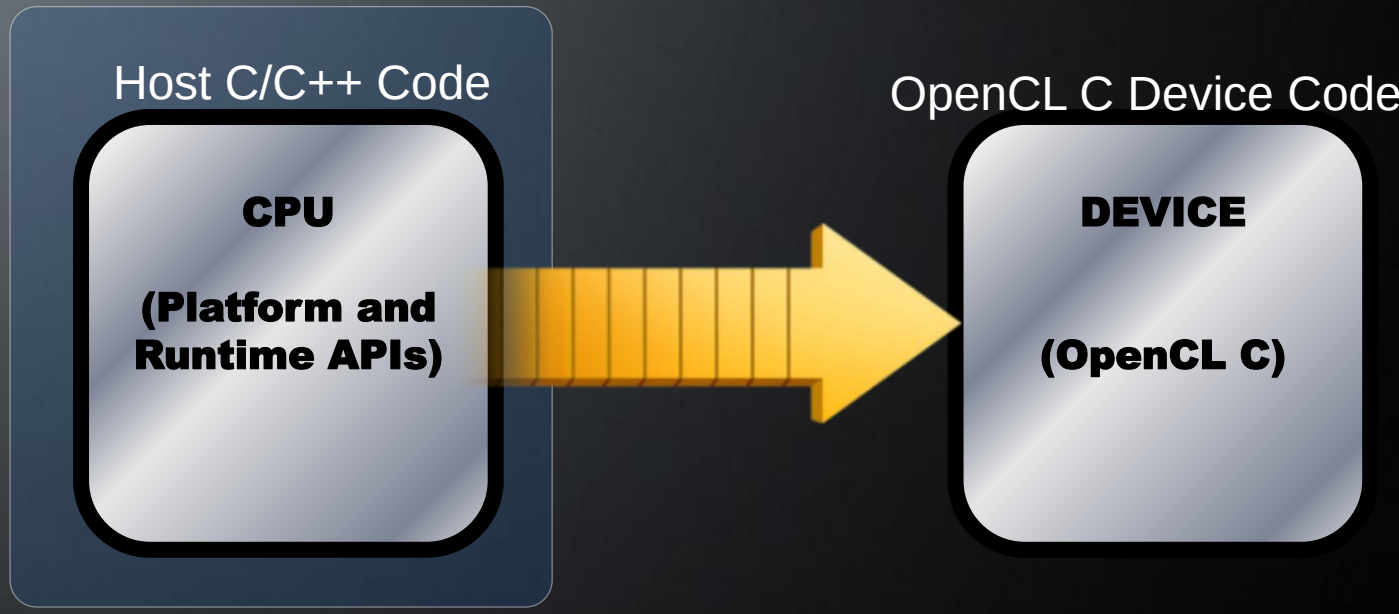
- Garrett is the Haskell expert and knows a lot more than I do about transforming Monads!

AGENDA

- Motivation
- Whistle stop introduction to OpenCL
- Bringing OpenCL to Haskell
- Lifting to something more in the spirit of Haskell
- Quasiquotation
- Issues we face using Haskell at AMD
- Questions

Motivation

OPENCL™ PROGRAM STRUCTURE



AMD

HELLO WORLD OPENCL™ C SOURCE

```
__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

```
__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);
```

HELLO WORLD - HOST PROGRAM

Define platform and queues

```
// create the OpenCL context on a GPU device

cl_context = clCreateContextFromType(0

// 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
```

Define Memory objects

```
// 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,
   &progra
```

Create the program

Build the program

```
//
er                                    , NULL, NULL,
   NULL);
```

Create and setup kernel

```
//
k                                              LL);

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

// 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);
```

Execute the kernel

```
// read output array
err = cl                                    , CL_TRUE,
   0, n*                                    ;
```

Read results on the host

AMD

USING HASKELL OUR GOAL IS TO WRITE THIS

```
import Language.OpenCL.Module

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

prog = initCL [$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 . putStr . map castCCharToChar . fst =<< readBuffer b 0 (hlen - 1)
```

USING HASKELL OUR GOAL IS TO WRITE THIS

```
import Language.OpenCL.Module

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

prog = initCL [$cl| Quasiquoting
      __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 platform $ \p ->
   using (bufferWithFlags hwlen [WriteOnly]) $ \b ->
      do [k] <- theKernels
         invoke k b `overRange` ([0], [hwlen], [1])
         liftIO . putStr . map castCCharToChar . fst =<< readBuffer b 0 (hlen - 1)
```

Monad transformers

Single source

OpenCL C code statically checked at compile time.

No OpenCL or Haskell compiler modifications

USING HASKELL OUR GOAL IS TO WRITE THIS

```
import Language.OpenCL.Module

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

prog = inCL [$cl| Quasiquoting
    __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 platform $ \p ->
  using (bufferWithFlags hwlen [WriteOnly]) $ \b ->
    do [k] <- theKernels
       invoke k b `overRange` ([0], [hwlen], [1])
       liftIO . putStr . map castCCharToChar . fst =<< readBuffer b 0 (hlen - 1)
```

Monad transformers

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

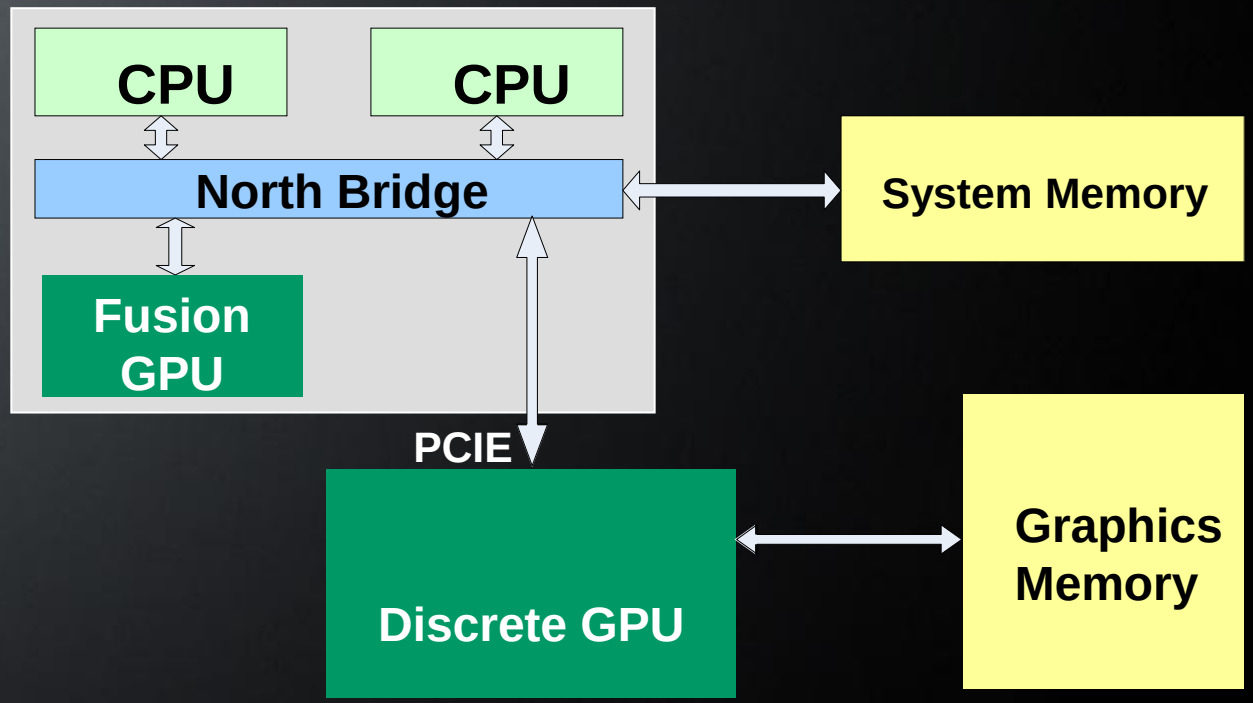
- In CUDA the default for runtime mode is:
  - Pick single device (always GPU)
  - All “kernels” in scope are exported to the host application for specific translation unit, i.e. calling kernels is syntactic and behave similar to static linkage.

Whistle stop introduction to OpenCL™

AMD

IT'S A HETEROGENEOUS WORLD

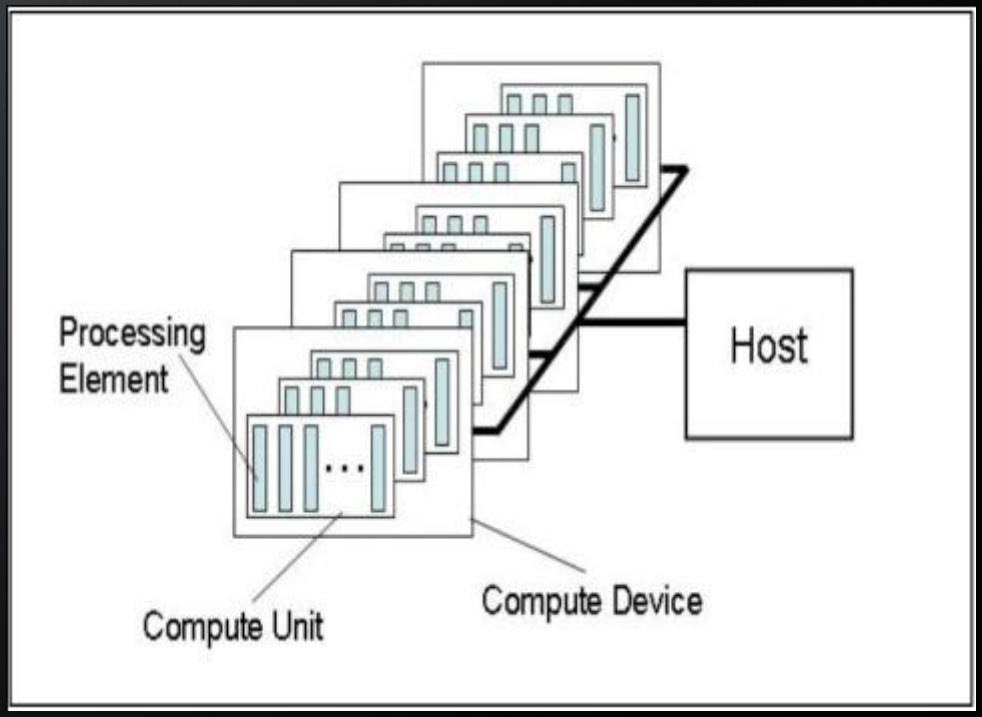
- A modern platform Includes:
  - One or more CPUs
  - One or more GPUs
  - And more



AMD

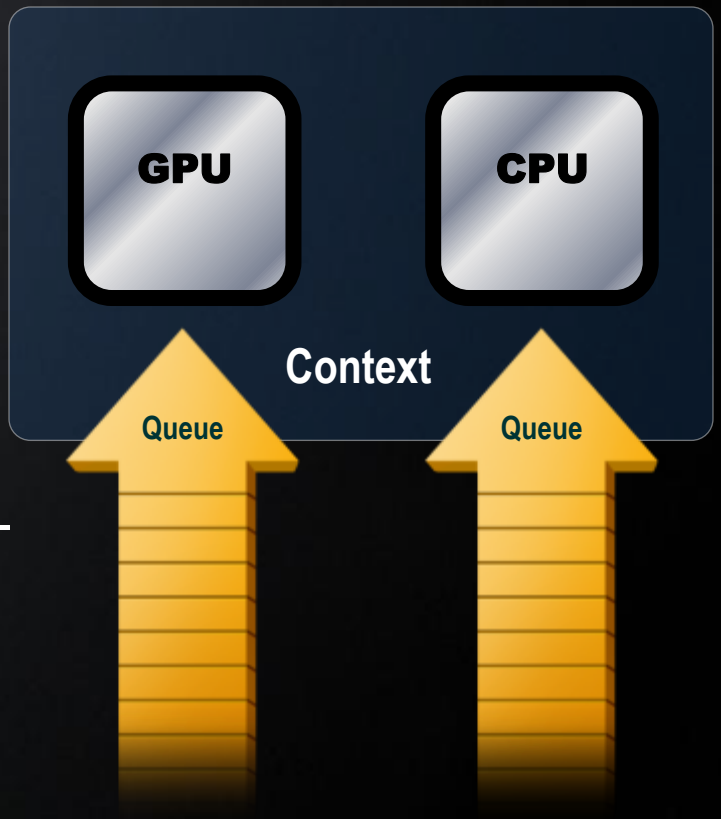
OPENCL™ PLATFORM MODEL

- One Host + one or more Compute Devices
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements



OPENCL™ EXECUTION MODEL

- An OpenCL application runs on a host which submits work to the compute devices
  - **Work item**: the basic unit of work on an OpenCL device
  - **Kernel**: the code for a work item. Basically a C function
  - **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
  - **Context**: The environment within which work-items executes … includes devices and their memories and command queues
- Applications queue kernel execution instances
  - Queued in-order … one queue to a device
  - Executed in-order or out-of-order



AMD

THE BIG IDEA BEHIND OPENCL™

- OpenCL execution model …
  - Define N-dimensional computation domain
  - Execute a kernel at each point in computation domain

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

THE BIG IDEA BEHIND OPENCL™

- OpenCL execution model …
  - Define N-dimensional computation domain
  - Execute a kernel at each point in computation domain

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

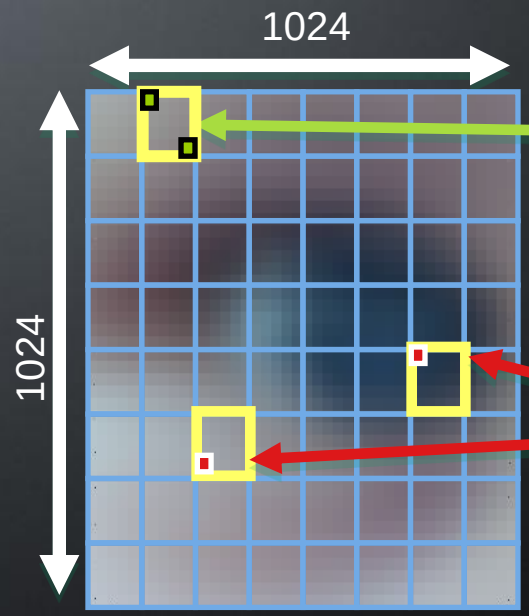
```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

AN N-DIMENSION DOMAIN OF WORK-ITEMS

- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (work group … executes together)



Synchronization between work-items possible only within workgroups
barriers and memory fences

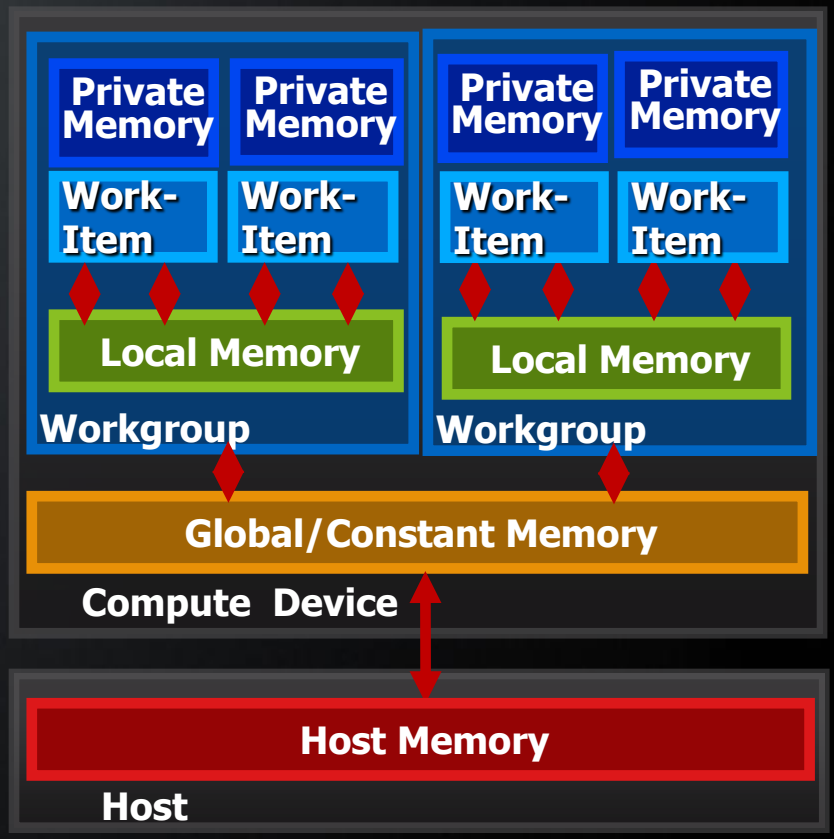
Cannot synchronize outside of a workgroup

- Choose the dimensions that are "best" for your algorithm

AMD

OPENCL™ MEMORY MODEL

- **Private Memory**
  - –Per work-item
- **Local Memory**
  - –Shared within a workgroup
- **Global/Constant Memory**
  - –Visible to all workgroups
- **Host Memory**
  - –On the CPU

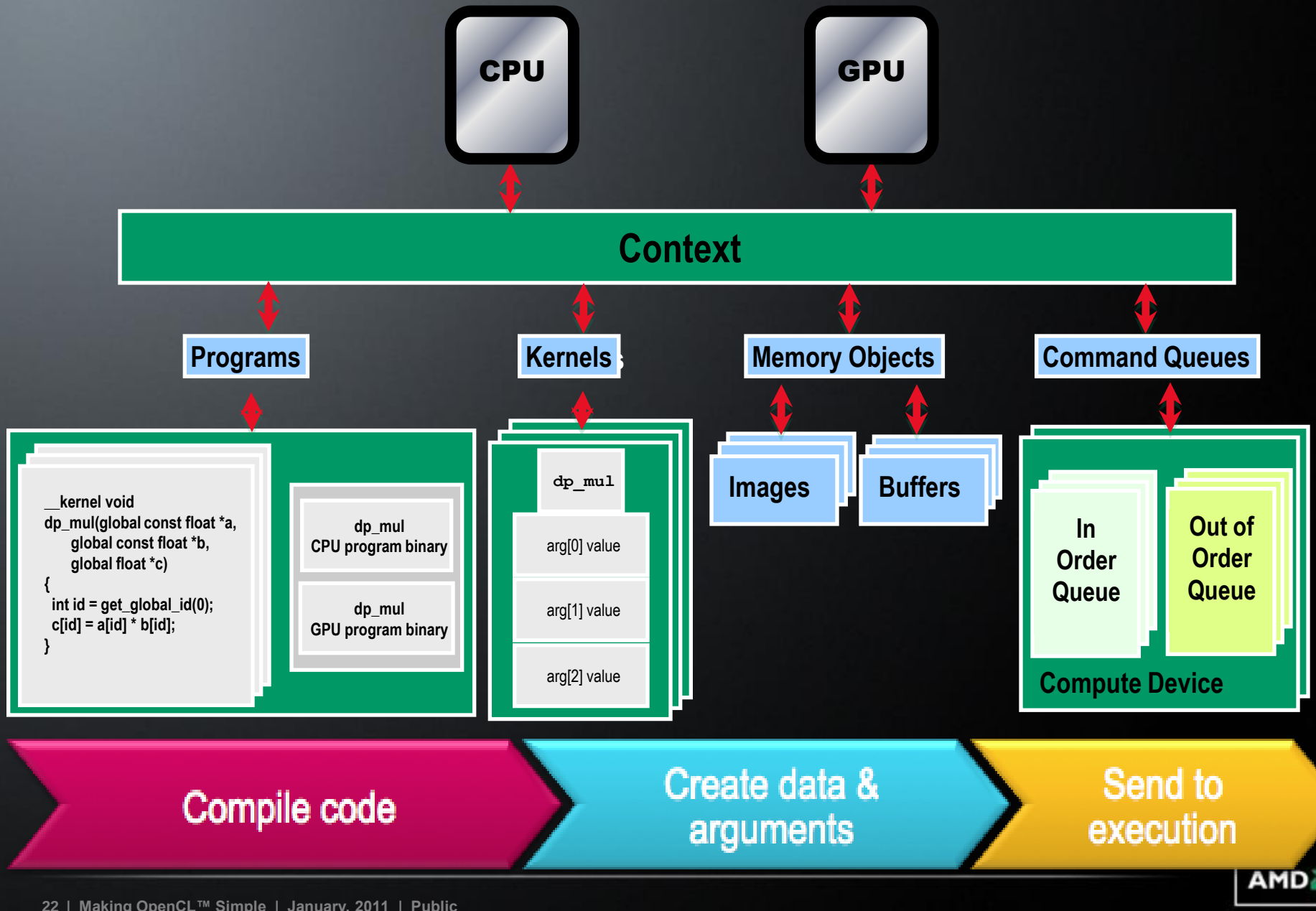


Memory management is Explicit

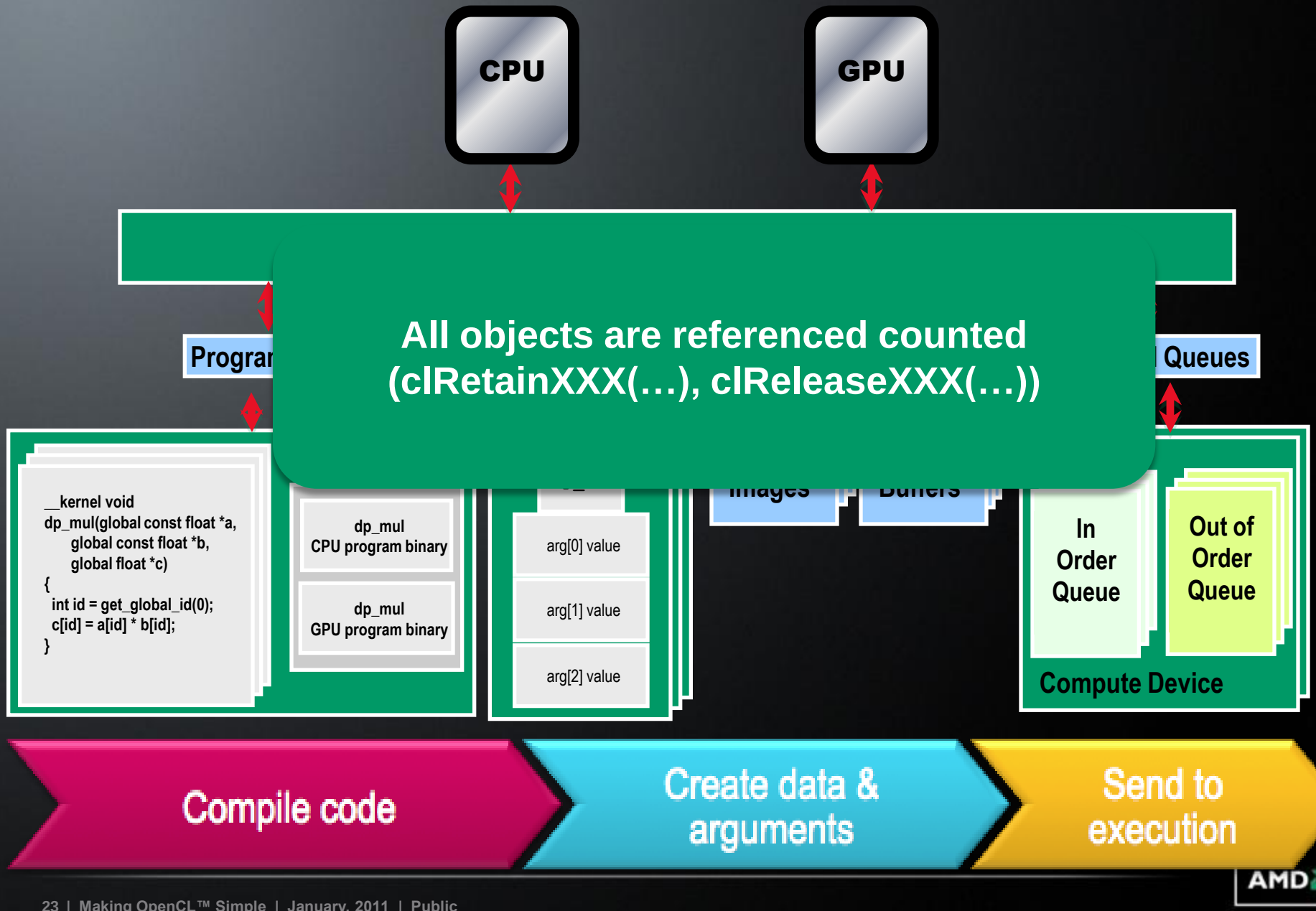
You must move data from host -> global -> local … *and* back

AMD

OPENCL™ SUMMARY



OPENCL™ SUMMARY



SYNCHRONIZATION: QUEUES & EVENTS

- Each individual queue can execute in-order or out-of-order
  - For in-order queue, all commands execute in order

- You must explicitly synchronize between queues
  - Multiple devices each have their own queue
  - Multiple queues per device
  - Use events to synchronize

- Events
  - Commands return events and obey waitlists
  - **clEnqueue***(. . . , **num_events_in_waitlist, *event_waitlist, *event**);

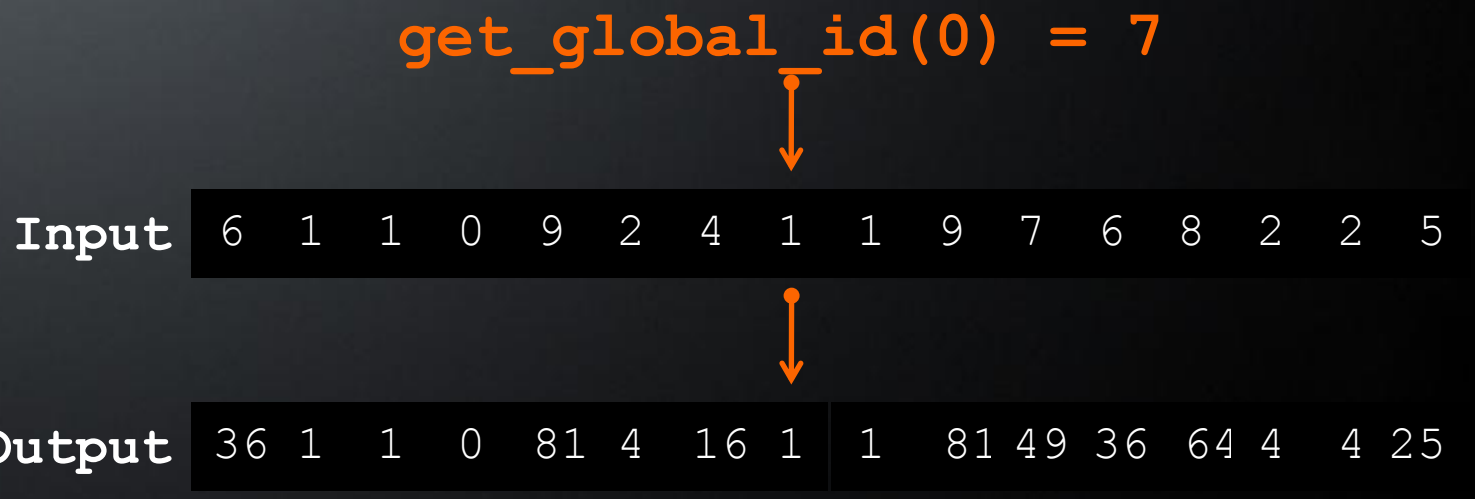
PROGRAMMING KERNELS: OPENCL™ C

- Derived from ISO C99
  - But without some C99 features such as standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- Language Features Added
  - Work-items and workgroups
  - Vector types
  - Synchronization
  - Address space qualifiers
- Also includes a large set of built-in functions
  - Image manipulation
  - Work-item manipulation,
  - Math functions, etc.

PROGRAMMING KERNELS: WHAT IS A KERNEL

- A data-parallel function executed by each work-item

```
kernel void square(global float* input,
                   global float* output)
{
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```



PROGRAMMING KERNELS: DATA TYPES

- Scalar data types
  - char , uchar, short, ushort, int, uint, long, ulong, float
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)
- Image types
  - image2d_t, image3d_t, sampler_t
- Vector data types
  - Vector lengths 2, 4, 8, & 16 (char2, ushort4, int8, float16, **double2**, …)
  - Endian safe
  - Aligned at vector length
  - Vector operations

Double is an optional type in OpenCL 1.0

Bringing OpenCL™ to Haskell

AMD

THE BASICS - HELLO WORLD

```
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] []
          putStr . map castCCharToChar =<<
                    enqueueBlockingReadBuffer q b 0 (hwlen - 1) []
```

MAPPING OPENCL™ STATIC VALUES

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

```
data Platform_
type Platform = Ptr Platform_

type CLPlatformInfo = Word32
```

```
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
```

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:

```
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

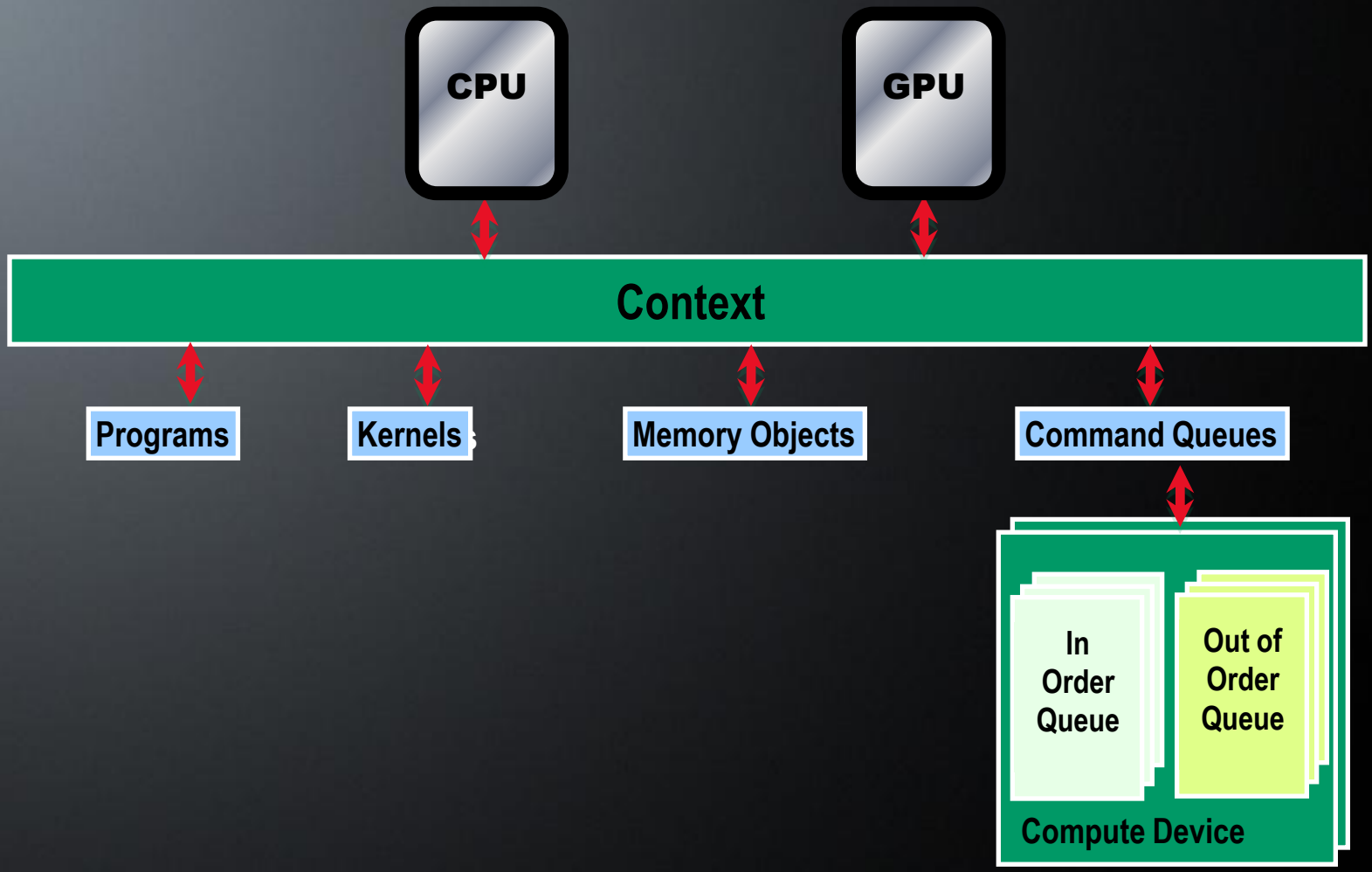
AMD

SO FAR NOTHING THAT INTERESTING

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

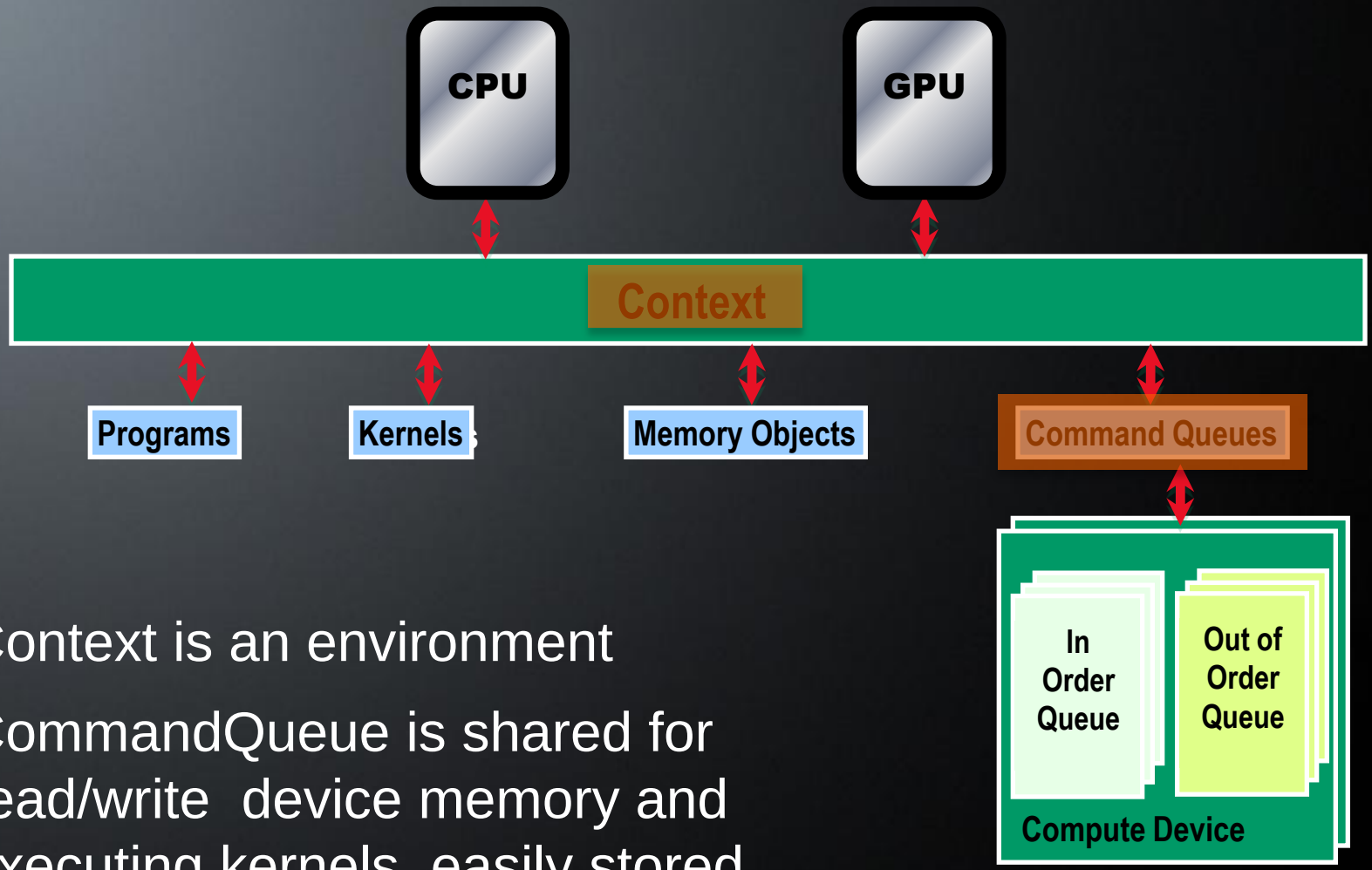
- HOpenCL Contextual API
  - Monad transformers to lift HOpenCL Native API beyond the IO monad
  - Quasiquotation to lift OpenCL source into Haskell as a first class citizen.

MOVE TO AN IMPLICIT MODEL



AMD

MOVE TO AN IMPLICIT MODEL



- 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:

```
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:

```
withNew :: MonadIO m => m Context -> ContextM t -> m t
```

- This mechanism is extensible: the quasiquotation 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.

LIFESPAN

- 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:

```
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

Quasiquotation

THE ROAD TO SINGLE SOURCE

- Embedded OpenCL C concrete syntax into Haskell with quasiquation

- Set of default qausiquoters:
  - [$cl| … |] OpenCL defult device, command queue and so on
  - [$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
- Antiquoting can be used to reference Haskell values
- Save OpenCL binaries for reloading at runtime (guarantees correctness)

HELLO WORLD REVISITED

```
import Language.OpenCL.Module

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

prog = initCL [$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 . putStr . 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)
```

REALLY WE WOULD LIKE TO WRITE

```
import Language.OpenCL.Module

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 . putStr . map castCCharToChar . fst =<< readBuffer b 0 (hlen - 1)
```

BUT THERE IS A PROBLEM (I THINK)

- Quasiquotation requires instances for Typeable and Data

- Remember that HOpenCL has definitions of the form:

```
data Platform_
type Platform = Ptr Platform_
```

- No way to derive Typeable and Data for Platform_

- This makes sense in the case that we want to build and retain values of type Platform at compile time, what does it mean to preserve a pointer?

WHAT WE REALLY WANT

- Build a delayed instance of CatchIO m that when evaluated at runtime performs OpenCL initialization
- Quasiquoting does not seem to allow this due to the requirement of Typeable and Data instances.
- Remember we do not want to build data values containing Platform_, rather we want to construct a function that will perform the act at runtime.
- Question: is there a known solution to this problem?

WE COULD THEN GENERATE KERNEL WRAPPERS

```
hello :: KernelInvocation r => Buffer Char -> r
hello = invoke …
```

or

```
hello :: Wraps t m n => [Char] -> n [Char]
hello = …
```

Issues we face using Haskell at AMD

USING HASKELL AT AMD

- Monads are a simple idea but they are not easy to explain and use in practice. At least for many people.
  - Originally looking at Garrett's Monad transformers I said it seemed complicated. His response was " you need to look at them sideways"! (He was right ☺)

- Haskell is not the Miranda that I learned as an undergraduate, it is big and a lot of very complicated type theory stuff. Really scary for many people!
  - Benjamin C. Pierce has a great talk discussing just this phenomena;
    - **Types Considered Harmful**, May 2008. Invited talk at *Mathematical Foundations of Programming Semantics (MFPS)."*
    - "I have long advocated type systems…but I've changed my mind"

USING HASKELL AT AMD

- Many Universities in the US do not teach Haskell!

- Perception of Haskell is often low:
  - "You probably know every Haskell programmer"
  - "Why not use Python or …"

- Finally, it is not easy to find interns with the required Haskell experience to work on these kinds of projects.

To be fair no one has said don't use Haskell!

QUESTIONS

DISCLAIMER

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. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. OpenCL is a trademark of Apple Inc. used with permission by Khronos. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.