

Device Fission Extensions for OpenCL™

Benedict R. Gaster | November 2010



OpenCL™ extensions

- KRH (e.g. cl_khr_d3d10_sharing)
 - Developed and approved by the OpenCL™ working group
 - Approved conformance tests
 - Included with OpenCL™ 1.1 conformance submission
 - Documentation on the Khronos OpenCL™ registry
- EXT (e.g. cl_ext_device_fission)
 - Developed by at least 2 or more members of the working group
 - No required conformance tests
 - Documentation shared on the Khronos OpenCL™ registry
- Vendor (e.g. cl_amd_printf)
 - Developed by a single vendor
 - Documentation still shared on the Khronos OpenCL™ registry



OpenCL™ Device Fission (cl_khr_device_fission)

- Provides an interface for sub-dividing an OpenCL™ device into multiple sub-devices
- Typically used to:
 - Reserve a part of the device for use for high-priority/latency-sensitive tasks
 - Provide more direct control for the assignment of work to individual compute units
 - Subdivide compute devices along some shared hardware feature like a cache
- Today supported by CPU and Cell Broadband devices
 - Multicore CPU devices (AMD and Intel)
 - IBM Cell Broadband
- In the future may we see support also for the GPU?
- Developed by
 - AMD, Apple, IBM, and Intel

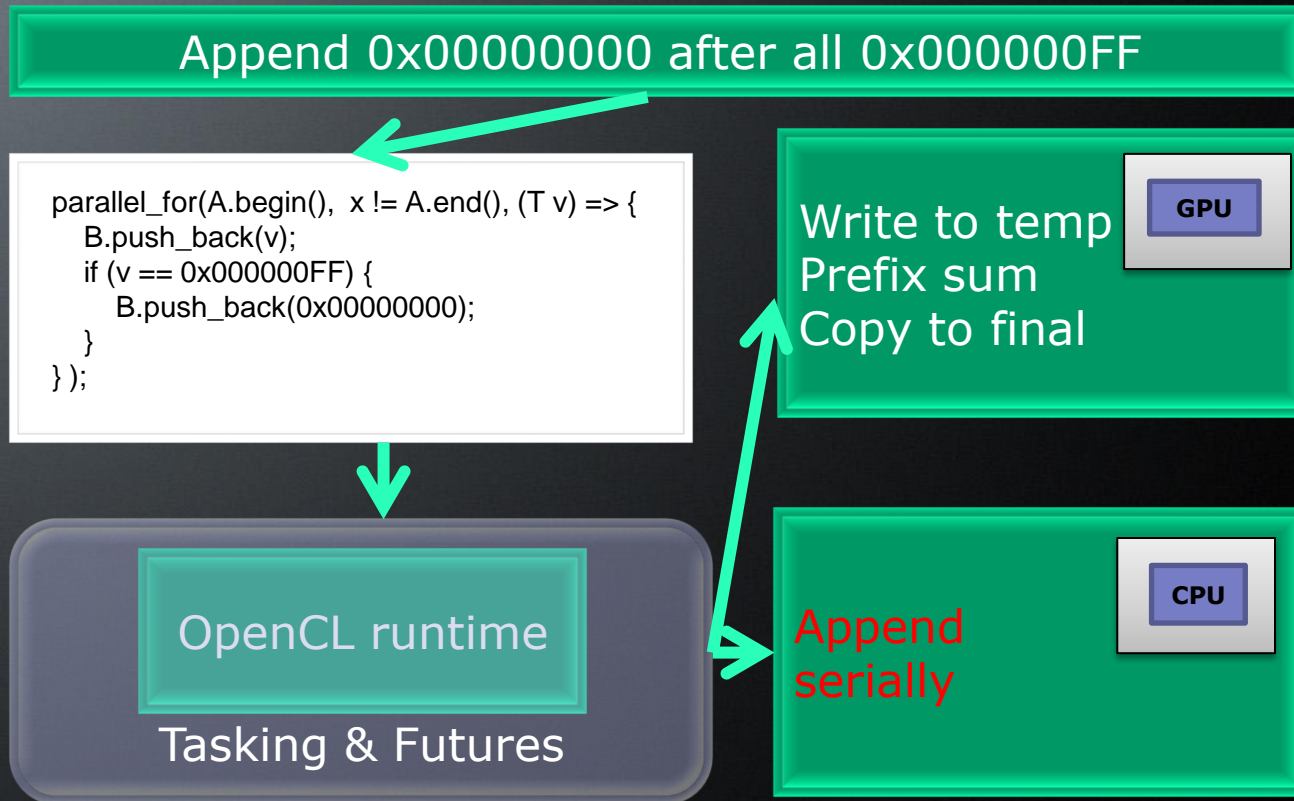


Parallel algorithms, containers, and tasks with OpenCL™

- Parallel algorithms
 - `parallel_for`
 - `parallel_reduction`
 - Asynchronous tasks with results (i.e. futures)
- Parallel containers
 - `vector` (including `push_back` and `insert`)
 - `queue` (blocking and unblocking)
 - `dictionary`
- For this talk we focus only on implementing *vector.push_back* on OpenCL™ CPU



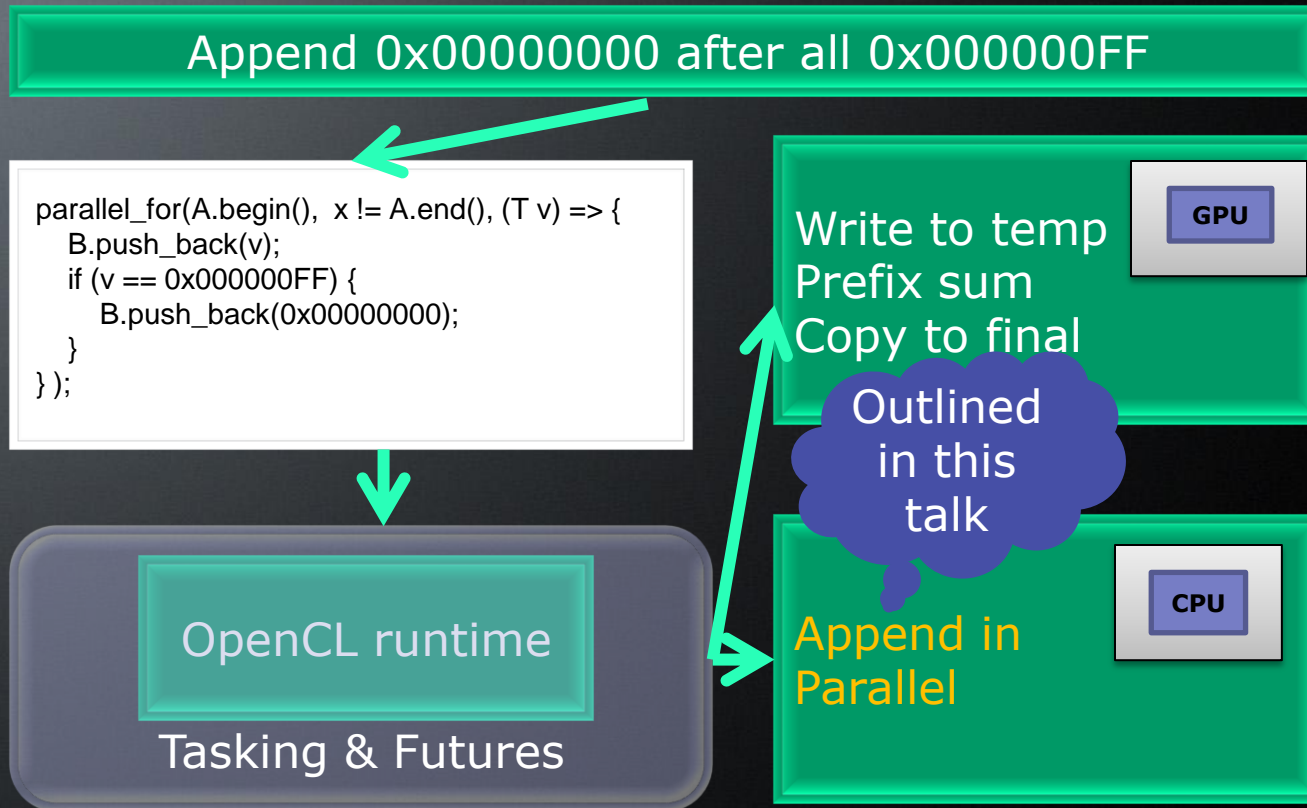
High-Level containers and parallel tasking with OpenCL™



Reference:
SPAP: A Programming Language for Heterogeneous
Many-Core Systems, Qiming Hou et al, 2010.



High-Level containers and parallel tasking with OpenCL™



Containers have sequential semantics

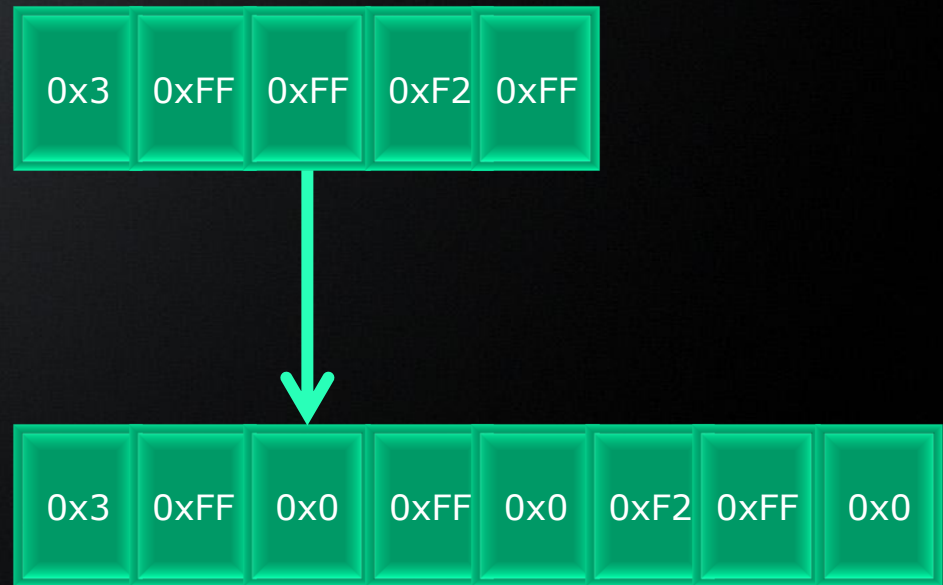
```
unsigned int padding( unsigned int n, unsigned int in[n],
                    unsigned int out[n*2])
{
    for (int i = 0, j = 0; i < n; i++, j++) {
        unsigned int x = in[i];
        out[j] = x;
        if (x == 0xFF) {
            out[++j] = 0x0;
        }
    }
}
```

- Focus on *push_back()* implementation
- Simple C implementation.



Containers have sequential semantics

```
unsigned int padding(unsigned int n, unsigned int in[n], unsigned int out[n*2])  
{  
    for (int i = 0, j = 0; i < n; i++, j++) {  
        unsigned int x = in[i];  
        out[j] = x;  
        if (x == 0xFF) {  
            out[++j] = 0x0;  
        }  
    }  
}
```



Naïve approach might simply replace the for with parallel_for

```
__kernel
void padding(
    __global uint * counter,
    __global uint * input,
    __global uint * output)
{
    uint x = input[get_global_id(0)];
    uint offset;
    if (x == 0xFF) {
        offset = atomic_add(counter, 2);
    }
    else {
        offset = atomic_inc(counter);
    }
    output[offset] = x;
    output[offset+1] = 0x0;
}
```



Naïve approach might simply replace the for with parallel_for

```
__kernel
void padding(
    __global uint * counter,
    __global uint * input,
    __global uint * output)
{
    uint x = input[get_global_id(0)];
    uint offset;
    if (x == 0xFF) {
        offset = atomic_add(counter, 2);
    }
    else {
        offset = atomic_inc(counter);
    }
    output[offset] = x;
    output[offset+1] = 0x0;
}
```

Problem:

Padding values are correctly inserted but there is no guarantee that original ordering will be preserved!



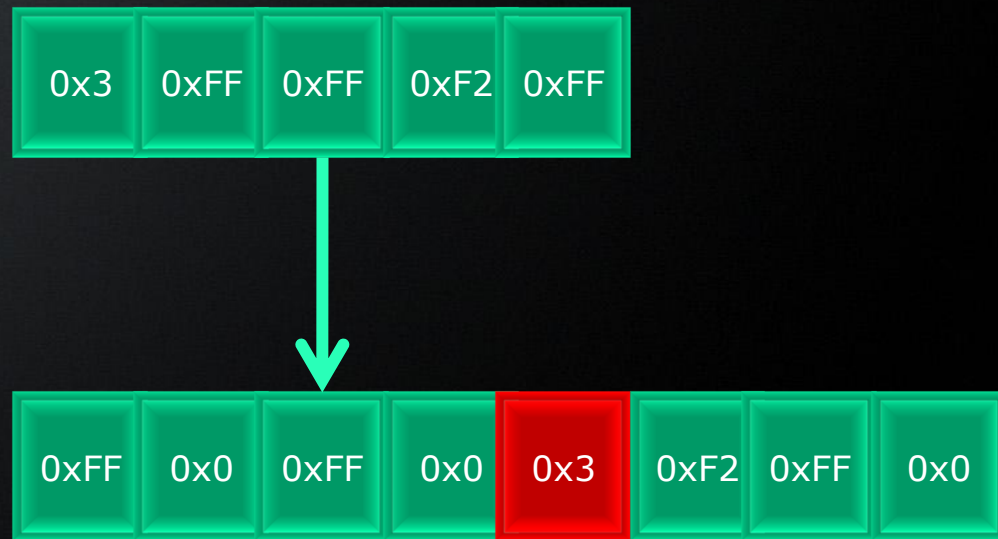
Naïve approach might simply replace the for with parallel_for

__kernel

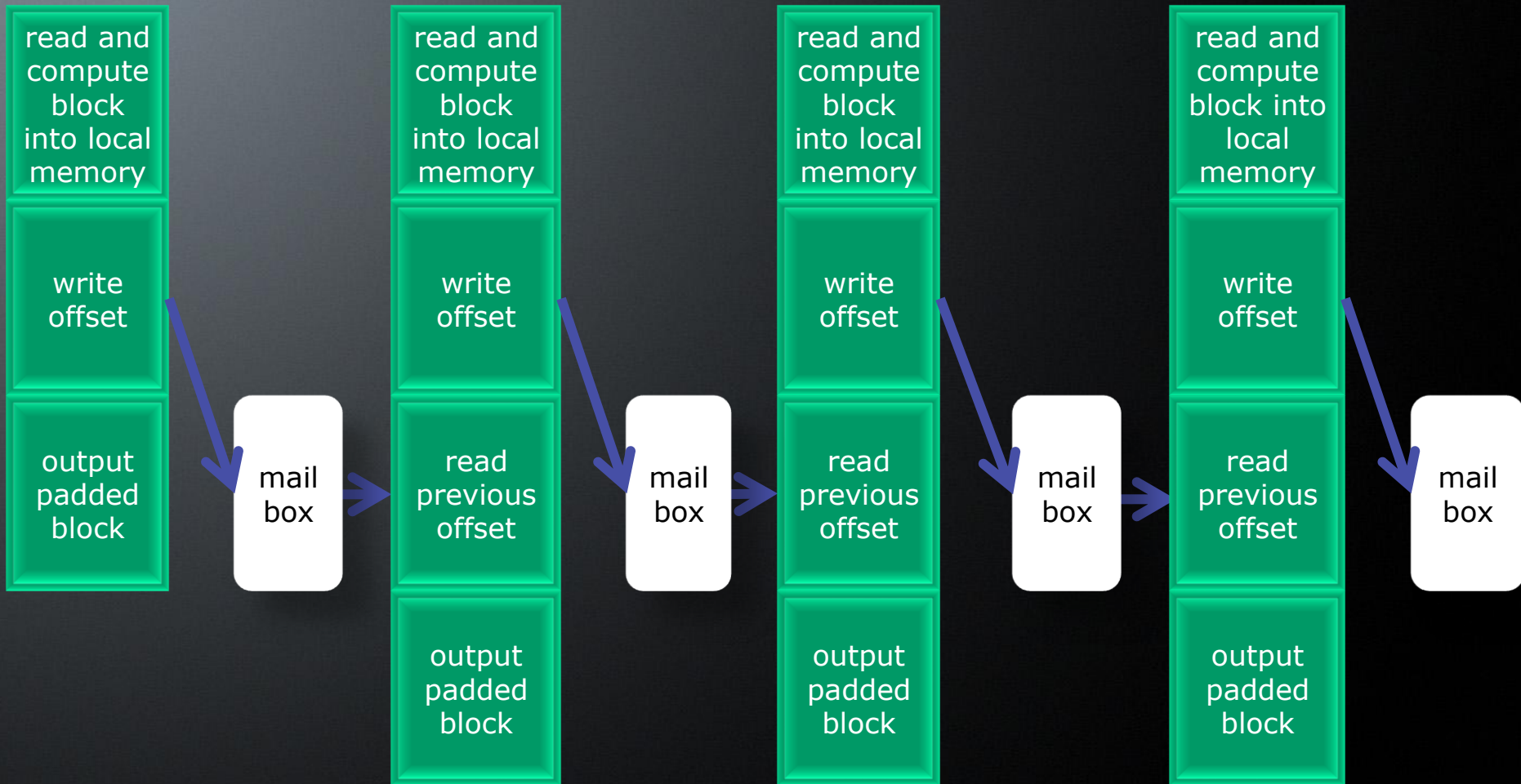
```
void insertPadding(  
    __global uint * counter,  
    __global uint * input,  
    __global uint * output)  
{  
    uint x = input[get_global_id(0)];  
    uint offset;  
    if (x == 0xFF) {  
        offset = atomic_add(counter, 2);  
    }  
    else {  
        offset = atomic_inc(counter);  
    }  
    output[offset] = x;  
    output[offset+1] = 0x0;  
}
```

Problem:

Padding values are correctly inserted but there is no guarantee that original ordering will be preserved



Recast as parallel pipeline pattern



Pipeline 1st attempt

```
__kernel void padding( __global uint * counter, __global uint * groupid, __global uint
* input, __global uint * output)
{
    uint x    = input[get_global_id(0)];
    size_t lid = get_local_id(0);
    size_t gid  = get_group_id(0);

    if (gid != 0) {
        if (lid == get_local_size(0) - 1) {
            while(1) {
                if (*groupid == (gid - 1)) {
                    break;
                }
            }
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
```



Pipeline 1st attempt cond

```
uint offset;
if (x == 0xFF) {
    offset = atomic_add(counter, 2);
    output[offset] = x;
    output[offset+1] = 0x0;
}
else {
    offset = atomic_inc(counter);
    output[offset] = x;
}

if (lid == get_local_size(0) - 1) {
    *groupid = gid;
}
}
```



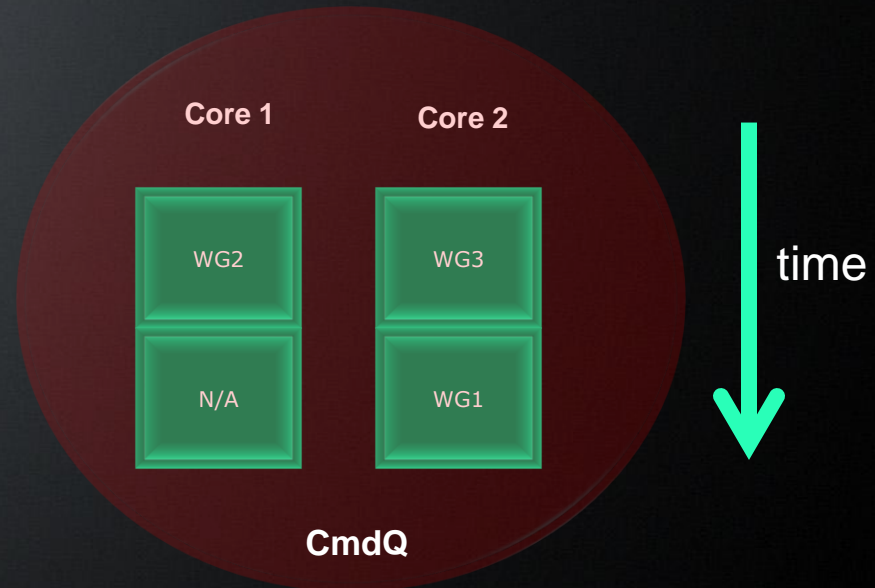
The problem with this approach

OpenCL™ makes no guarantee about work-group execution order

Problem:

Native approach has no guarantee of progress

Note, this is not resolved by issuing only two work-groups



Device Fission can guarantee progress

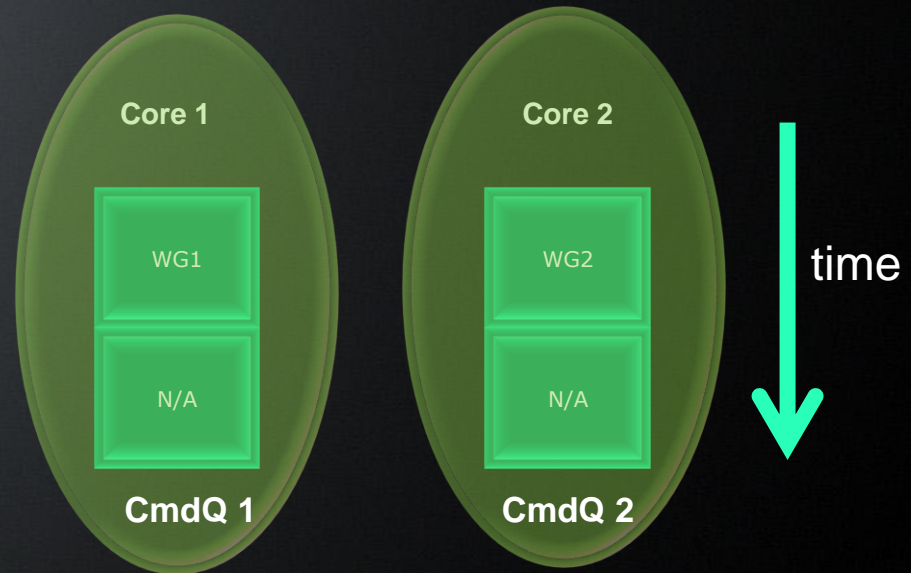
Solution:

Multiple devices guarantee that each will make progress independent of the other

Each device must have a corresponding command queue

Launch one work-group per command queue (often referred to as fill the machine)

Use native kernels to manually control “work-group” configuration



Native Kernels

- Enqueue C/C++ functions, compiled by the host compiler, to execute from within an OpenCL™ command queue

```
cl_int clEnqueueNativeKernel (cl_command_queue command_queue,  
                             void (*user_func)(void *)  
                             void *args,  
                             size_t cb_args,  
                             cl_uint num_mem_objects,  
                             const cl_mem *mem_list,  
                             const void **args_mem_loc,  
                             cl_uint num_events_in_wait_list,  
                             const cl_event *event_wait_list,  
                             cl_event *event)
```

- There is no guarantee that the function will execute in same thread that the enqueue was performed; must be careful about thread-local-storage usage



Pipeline 2nd attempt (host device fission code)

```
std::vector<cl::Platform> platforms;  
cl::Platform::get(&platforms);  
if (platforms.size() == 0) {  
    // handle error case  
}  
cl_context_properties properties[] = { CL_CONTEXT_PLATFORM,  
(cl_context_properties)(platforms[0])(), 0};  
  
cl::Context Context(CL_DEVICE_TYPE_CPU, properties);  
  
std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();  
  
if (devices[0].getInfo<CL_DEVICE_EXTENSIONS>().find("cl_ext_device_fission")  
== std::string::npos) {  
    // handle case when fission not supported, e.g. fall back to sequential version  
}
```



Pipeline 2nd attempt (host device fission code)

```
cl_device_partition_property_ext subDeviceProperties[] =  
    { CL_DEVICE_PARTITION_EQUALLY_EXT, 1, CL_PROPERTIES_LIST_END_EXT,  
0};
```

```
devices[0].createSubDevices(subDeviceProperties, &subDevices);  
if (subDevices.size() <= 0) {  
    // handle error case  
}
```

```
counterV = new cl_uint[subDevices.size()]; // mailboxes
```

```
unsigned int j = 0;  
for (std::vector<cl::Device>::iterator i = subDevices.begin(); i !=  
subDevices.end(); i++) {  
    queues.push_back(cl::CommandQueue(context, *i));  
    counterV[j++] = 0;  
}
```

```
// Code to allocate and setup buffers and so on
```



Pipeline 2nd attempt (host device fission code) cond

```
cl_uint * args[6] = { NULL, NULL, NULL, NULL, reinterpret_cast<cl_uint  
*>(size/numComputeUnits), 0x0 };
```

```
std::vector<cl::Memory> memArgs;  
memArgs.push_back(input);  
memArgs.push_back(output);  
memArgs.push_back(counter);  
memArgs.push_back(blocks[0]);
```

```
std::vector<const void *> memLocations;  
memLocations.push_back(&args[0]);  
memLocations.push_back(&args[1]);  
memLocations.push_back(&args[2]);  
memLocations.push_back(&args[3]);
```

```
unsigned int groupID = 0;
```

```
std::vector<cl::Event> events;
```



Pipeline 2nd attempt (host device fission code) cond

```
groupID = 0;
cl::Event event;
for (unsigned int i = 0; i < numComputeUnits; i++) {
    memArgs.pop_back();
    memArgs.push_back(blocks[i]);
    args[5] = reinterpret_cast<cl_uint *>(groupID);
    queues[i].enqueueNativeKernel(
        padding,
        std::make_pair(static_cast<void *>(args), sizeof(cl_uint)*arraySize(args)),
        &memArgs,
        &memLocations,
        NULL,
        &event);
    events.push_back(event);
    groupID++;
}
cl::Event::waitForEvents(events);
```



Pipeline 2nd attempt (native function)

```
void padding(void * args)
{
    unsigned int ** argsPtr = static_cast<unsigned int **>(args);
    unsigned int * input     = *argsPtr++;
    unsigned int * output    = *argsPtr++;
    unsigned int * counter   = *argsPtr++;
    unsigned int * localBlock = *argsPtr++;

    unsigned int  blockSize = reinterpret_cast<unsigned int>(*argsPtr++);
    unsigned int  groupID   = reinterpret_cast<unsigned int>(*argsPtr);

    unsigned int  offset    = groupID * blockSize;
    unsigned int  localBlockSize = 0;
```



Pipeline 2nd attempt (native function) cond

```
for (unsigned int i = offset; i < blockSize+offset; i++, localBlockSize++) {
    unsigned int x = input[i];
    localBlock[localBlockSize] = x;
    if (x == 0x000000FF) {
        localBlock[++localBlockSize] = 0x0;
    }
}
if (groupID > 0) {
    offset = counter[groupID-1];
    while (offset == 0)
        offset = counter[groupID-1]; // read mailbox
}
counter[groupID] = localBlockSize+offset; // write to mailbox

for (unsigned int i = offset, j = 0; i < localBlockSize+offset; i++, j++) {
    output[i] = localBlock[j];
}
}
```



A lot more can still be done...

- Presented implementation is not optimized for CPU cache usage
 - Divide work into L1-D cache size blocks of work
 - Extend mailbox scheme to support multiple iterations, so
 - Core 1 – $B_0, \dots, B_{\#cores}, \dots, B_{\#cores*2}$
 - Core 2 – $B_1, \dots, B_{\#cores+1}, \dots, B_{(\#cores*2)+1}$
 - ...
- Presented implementation does not use blocking mailboxes
 - Statically sized queues
 - Block when writing to a full queue (not needed for this algorithm)
 - Block when reading from an empty queue
- Avoid using native kernels (or even device fission)?
 - Directly through support for global work-group synchronization
 - OpenCL™ statically sized queues (as above), that use global work-group synchronization underneath



Questions and Answers

Visit the OpenCL Zone on developer.amd.com

<http://developer.amd.com/zones/OpenCLZone/>

- Tutorials, developer guides, and more
- OpenCL Programming Webinars page includes:
 - Schedule of upcoming webinars
 - On-demand versions of this and past webinars
 - Slide decks of this and past webinars



Disclaimer and Attribution

DISCLAIMER

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

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.

ATTRIBUTION

© 2010 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, ATI, the ATI logo, AMD Opteron, Radeon, FirePro, FireStream and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft, Windows, Windows Vista, and DirectX™ are registered trademarks of Microsoft Corporation in the United States and/or other jurisdictions. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. Other names are for informational purposes only and may be trademarks of their respective owners.

