Channels
Or why the managed_ptr is more complex than it appears

Ruyman Reyes Castro, Principal Software Engineer, Programming Models
SYCL Buffers

• In SYCL, buffers represent allocation of memory on the system
  – The user has no control on where the allocation resides

• Data follows execution across devices on the system
  – User can provide hints to where data will be
  – Dataflow patterns can be extracted to optimize performance

• Data cannot be extracted from buffers directly, accessors are used to indicate where access is requested

```cpp
buffer<float, 1, CustomAllocator> buf{myPtr, range<1>{1}};
```
SYCL Buffers – How do they work

• A buffer holds a directory of different copies of the data in different OpenCL contexts and places in host memory
 • Last place where data was accessed holds most updated data
 • When data is required on different context/host, is moved across the heterogeneous system
 • Data is updated using the most efficient method for the platform
Traditional views of memory

- Can allocate memory, use it on the CPU
Slightly more complex...

Multi-CPU system

- Can allocate memory anywhere
- Can use it anywhere
- Access time may not be uniform! (NUMA)

Fortran

C++

Custom Allocators
Separate memory layouts

Host-directed accelerator model:

- Data is off-loaded on the device
- Host allocates on device
- No mapped pointers
Partially accessible pointers

Host-directed model

- Data is off-loaded on the device
- Host allocates on device
- Mapped pointer access device on host
The illusion of memory

Virtual Shared memory

- Illusion of coherent access, performance impact
- Special malloc function
- System handles transparently access in host and accelerator
- No atomics or concurrent access across devices
What we all ideally want

- Device an accelerator share physical memory
- Atomic operations are possible in all levels
- Hardware complexity is much higher

Real shared memory access
Implementing the SYCL buffer

• When data is required in a different context, we need to open a channel from the previous one to the new one.

• This channel represents the fastest way of communicating SYCL contexts.

• The actor can be either in the new context or in the old one.
  - The new context can get the information from the previous one.
  - The old context can put information on the new one.
Is a SYCL buffer the right abstraction for C++?

- SYCL buffers have some limitations
  - Group Working on improvements for next specification

- SYCL implementations shipped to an specific system, implementor knows all possible connections between OpenCL contexts or devices

- Is this the case in C++?
  - Offering a generic managed_ptr would need each implementor to provide its own implementation or customization point
  - Some vendors or libraries may implement optimized channels for execution for a certain platform, how do they integrate their solutions to work with the managed_ptr?
The Channel interface

- A channel is a simple interface, defines:
  - An asynchronous put method to put data on a channel
  - A blocking get method that gets data from the channel
- The get method returns an object that has access to some portion of memory in the channel
  - Only one side of the channel can access a **locked_page**

```cpp
/** Channel.
 * Generic Channel interface
 */

template<class ChannelT>
class Channel {

public:
  Channel() = default;
  Channel(const Channel&) = default;
  Channel(Channel&&) = default;
  ~Channel() = default;

  // Put
template<typename U>
  void put(t off t off, size t nElems, U * ptr) = delete;

  // Get
template<typename T>
  locked page<T> get(t off t offset, size t nElems) = delete;
};

using LocalChannel = Channel<channels::Local>;
using MPIChannel = Channel<channels::MPI>;
```
A trivial example using Threads

```cpp
using nbsd::concurrent::ThreadPool;
ThreadPool tp;
{
    LocalChannel c(500ul);

    size_t nElems = 100ul;
    // Initialization of memory
    {
        auto p = c.get<float>({0, nElems});
        for (size_t i = 0; i < nElems; i++) {
            p.get()[i] = 0.0f;
        }
    }

    // We initialize each element of the memory
    // to its position,
    // but we do it in chunks of chunkSize.
    size_t chunkSize = 10u;
    size_t numChunks = nElems / chunkSize;
    for (size_t cId = 0; cId < numChunks; cId++) {
        tp.AddJob([=,&c](){
            get example(std::thread::get id(), cId, chunkSize, c);
        });
    }
}
```

```cpp
// Channel<ThreadExecutor>

// get example(std::thread::id pid, size_t start, size_t chunkSize, LocalChannel& c) {
{
    // For the duration of this block, no other thread can access the channel
    auto p = c.get<float>(start * chunkSize, chunkSize);
    #ifdef VERBOSE
    std::cout << " pid: " << pid << " start:
    << start << " chunkSize " << chunkSize << std::endl;
    #endif // VERBOSE
    for (size_t i = 0; i < chunkSize; i++) {
        p.get()[i] = start * chunkSize + i;
    }
    
    // put example(std::thread::id pid, size_t i, LocalChannel& c) {
    {
        // Every thread can put its value on the channel, no need for sync
        float val = 3.0;
        c.put<float>({1, 1, &val});
    }
```
Using Execution Contexts

- Execution agents from a given Execution Context can obtain an allocator from the Execution Context.
- In order for an execution agent to access memory from a different Execution Context, a Channel is required.
- Custom implementations for pairs of Execution Contexts can be provided.
- **Developers can implement their own Channels for two given Execution Context**
  - This facilitates the creation of third-party libraries.
- **Not required if your system is fully coherent**
  - But even if it is, developers can create a channel to connect a third-party device.
Thanks for Listening

@codeplaysoft  info@codeplay.com  codeplay.com