Task graphs in SYCL and concurrent execution in triSYCL

HiPEAC 2016 SYCL tutorial

Ronan Keryell

Xilinx Research Labs

2016/01/18

Outline

Outline

C++14

- 2 Open Source compilers available *before* ratification (GCC & Clang/LLVM)
- Confirm new momentum & pace: 1 major (C++11) and 1 minor (C++14) version on a 6-year cycle
- Next big version expected in 2017 (C++1z)
  - Already being implemented! ☺
- Monolithic committee replaced by many smaller *parallel* task forces
  - Parallelism TS (Technical Specification) with Parallel STL
  - Concurrency TS (threads, mutex...)
  - Array TS (multidimensional arrays *à la* Fortran)
  - Transactional Memory TS...

  Race to parallelism! Definitely matters for HPC and heterogeneous computing!

C++ is a complete new language

- Forget about C++98, C++03...
- Send your proposals and get involved in C++ committee (pushing heterogeneous computing)!

Modern C++ & HPC (I)

- Huge library improvements
  - <thread> library and multithread memory model <atomic> $\leadsto$ HPC
  - Hash-map
  - Algorithms
  - Random numbers
  - ...
- Uniform initialization and range-based for loop

```
std::vector<int> my_vector { 1, 2, 3, 4, 5 };
for (int &e : my_vector)
  e += 1;
```

- Easy functional programming style with λ expressions (anonymous functions)

```
std::transform(std::begin(v), std::end(v), [] (int e) { return 2*e; });
```

Modern C++ & HPC (II)

- Lot of meta-programming improvements to make meta-programming ~~easy~~ easier: variadic templates, type traits <type_traits>...
- Make simple things simpler to be able to write generic numerical libraries, etc.
- Automatic type inference for terse programming
  - Python 3.x (interpreted):

```python
def add(x, y):
  return x + y
print(add(2, 3))     # 5
print(add("2", "3")) # 23
```

  - Same in C++14 but compiled + static compile-time type-checking:

```cpp
auto add = [] (auto x, auto y) { return x + y; };
std::cout << add(2, 3) << std::endl;         // 5
std::cout << add("2"s, "3"s) << std::endl;  // 23
```

Without using templated code! ~~`template <typename >`~~ ☺

Modern C++ & HPC (III)

- R-value references & std :: move semantics
  - matrix_A = matrix_B + matrix_C
    - Avoid copying (TB, PB, EB... ☹) when assigning or function return
- Avoid raw pointers, `malloc()/free()/delete[]`: use references and smart pointers instead

```
// Allocate a double with new() and wrap it in a smart pointer
auto gen() { return std::make_shared<double> { 3.14 }; }
[...]
{
  auto p = gen(), q = p;
  *q = 2.718;
  // Out of scope, no longer use of the memory: deallocation happens here
}
```

Modern C++ & HPC *(IV)*

- C++14 generalizes constexpr to statements

```
constexpr auto fibonacci(int v) {
  long long int u_n_minus_1 = 0;
  auto u_n = u_n_minus_1 + 1;
  for (int i = 1; i < v; ++i) {
    auto tmp = u_n;
    u_n += u_n_minus_1;
    u_n_minus_1 = tmp;
  }
  return u_n;
}
int main() {
  constexpr auto result = fibonacci(80);
  std::cout << result << std::endl;
  return 0;
}
```

Modern C++ & HPC (V)

Compiled to

```
movabsq $23416728348467685, %rsi # imm = 0x533163EF0321E5
movl    $_ZSt4cout, %edi
callq   _ZNSo9_M_insertIxEERSoT_
```

- Lot of other amazing stuff...
- Allow both low-level & high-level programming... Useful for heterogeneous computing

Outline

C++11 std::thread

- It's all in the standard! http://en.cppreference.com/w/cpp/thread/thread/thread

```
#include <chrono>
#include <iostream>
#include <thread>

void f(int n) {
  std::cout << "Thread " << n << "executing" << std::endl;
  std::this_thread::sleep_for(std::chrono::milliseconds(100));
}

  // Launch f in thread t:
  std::thread t(f, 1);
  // The same with a lambda
  std::thread t2([] {std::cout << "Hello!" << std::endl;});
```

- ∃ higher-level constructs: std::async, std::future/std::promise

C++11 std::async

- Function objects that *may* be executed in parallel by some thread pool

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>

template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end) {
  auto len = end - beg;
  if (len < 1000)
    // Small enough for quicker sequential execution
    return std::accumulate(beg, end, 0);

  // Divide and conquer the world!
  RAIter mid = beg + len/2;
  // This returns a std::future
  auto handle = std::async(std::launch::async,
                           parallel_sum<RAIter>, mid, end);
  int sum = parallel_sum(beg, mid);
  // Get the std::future value from the inner std::promise
  return sum + handle.get();
}

int main() {
  // 10000 elements initialized with 1
  std::vector<int> v(10000, 1);
  std::cout << "sum = "
            << parallel_sum(v.begin(), v.end()) << std::endl;
}
```

- std::future/std::promise can transfer exceptions across threads too

http://en.cppreference.com/w/cpp/thread/async

Example of C++ library: TBB

- Open Source library started by Intel
- Threads + work-stealing scheduling
- Containers, algorithms, scalable allocators
- Data-flow graph with message queues

Example of C++ extensions: OpenMP (I)

- Single source #pragma extensions
- Support C, C++, Fortran with same #pragma
  - Other unofficial standards (Pythran for Python...)
- Task-graph oriented programming
- Multithread SMP support
- Loop parallelization
- Relaxed memory consistency model
- SIMD vectorization
- Accelerator offloading with hierarchical parallelism
- Transactional memory (optimistic speculative execution)
- Runtime API (environment queries, locks)

Example of C++ extensions: OpenMP (II)

```
int fib(int n) {
  int i, j;
  if (n<2)
    return n;
  else {
#pragma omp task shared(i)
    i = fib(n-1);
#pragma omp task shared(j)
    j = fib(n-2);
#pragma omp taskwait
    return i+j;
  }
}
```

Outline

What about heterogenous computing???

- C++ std :: thread is great...
- ...but supposed shared unified memory (SMP) ☹
  - What if accelerator with own separate memory?
  - What if using distributed memory multi-processor system (MPI...)?
- ⤳ Extend the concepts...
  - Replace raw unified-memory with buffer objects
  - Define with accessor objects which/how buffers are used
  - Since accessors are already here to define dependencies, no longer need for std :: future/std :: promise! ☺
  - Add concept of queue to express where to run the task
  - Also add all goodies for massively parallel accelerators (OpenCL/Vulkan/SPIR-V) in clean C++

SYCL ≡ pure C++14 DSEL

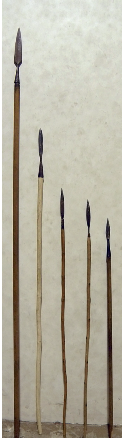
- Implement concepts useful for heterogeneous computing
- Asynchronous task graph
- Buffers to define location-independent storage
- Accessors to express usage for buffers and pipes: read/write/...
- Hierarchical parallelism
- Hierarchical storage
- Single source programming model
  - Take advantage of CUDA & OpenMP simplicity and power
  - Compiled for host *and* device(s)
  - Enabling the creation of C++ higher level programming models & C++ templated libraries
- Most modern C++ features available for OpenCL
  - Programming interface based on abstraction of OpenCL components (data management, error handling...)
  - Provide OpenCL interoperability
- Host fallback (debug and symmetry for SIMD/multithread on host)
- Directly executable DSEL
- Host emulation for free & no compiler needed for experimenting

Puns and pronunciation explained

OpenCL SYCL

sickle ['si-kəl]

OpenCL SPIR



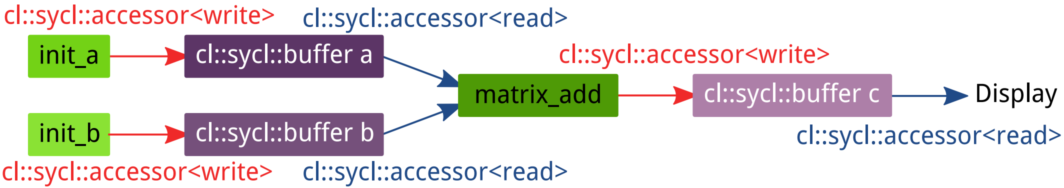
spear ['spir]

Complete example of matrix addition in OpenCL SYCL

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

constexpr size_t N = 2;
constexpr size_t M = 3;
using Matrix = float[N][M];

int main() {
 Matrix a = { { 1, 2, 3 }, { 4, 5, 6 } };
 Matrix b = { { 2, 3, 4 }, { 5, 6, 7 } };

 Matrix c;

 {// Create a queue to work on default device
  queue myQueue;
  // Wrap some buffers around our data
  buffer<float, 2> A { a, range<2> { N, M } };
  buffer<float, 2> B { b, range<2> { N, M } };
  buffer<float, 2> C { c, range<2> { N, M } };
  // Enqueue some computation kernel task
  myQueue.submit([&](handler& cgh) {
   // Define the data used/produced
   auto ka = A.get_access<access::read>(cgh);
   auto kb = B.get_access<access::read>(cgh);
   auto kc = C.get_access<access::write>(cgh);
   // Create & call OpenCL kernel named "mat_add"
   cgh.parallel_for<class mat_add>(range<2> { N, M },
     [=](id<2> i) { kc[i] = ka[i] + kb[i]; }
   );
  }); // End of our commands for this queue
 } // End scope, so wait for the queue to complete.
   // Copy back the buffer data with RAII behaviour.
 return 0;
}
```

Asynchronous task graph model

- Change example with initialization kernels instead of host
- Theoretical graph of an application described *implicitly* with kernel tasks using buffers through accessors



- Possible schedule by SYCL runtime:

⤳ Automatic overlap of kernels & communications

- Even better when looping around in an application
- Assume it will be translated into pure OpenCL event graph
- Runtime uses as many threads & OpenCL queues as necessary (GPU synchronous queues, AMD compute rings, AMD DMA rings...)

Task graph programming — the code

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;
int main() {
  { // By sticking all the SYCL work in a {} block, we ensure
    // all SYCL tasks must complete before exiting the block

    // Create a queue to work on
    queue myQueue;
    // Create some 2D buffers of float for our matrices
    buffer<double, 2> a({ N, M });
    buffer<double, 2> b({ N, M });
    buffer<double, 2> c({ N, M });
    // Launch a first asynchronous kernel to initialize a
    myQueue.submit([&](auto &cgh) {
        // The kernel write a, so get a write accessor on it
        auto A = a.get_access<access::write>(cgh);

        // Enqueue parallel kernel on a N*M 2D iteration space
        cgh.parallel_for<class init_a>({ N, M },
                            [=] (auto index) {
                              A[index] = index[0]*2 + index[1];
                            });
      });
    // Launch an asynchronous kernel to initialize b
    myQueue.submit([&](auto &cgh) {
        // The kernel write b, so get a write accessor on it
        auto B = b.get_access<access::write>(cgh);
        /* From the access pattern above, the SYCL runtime detect
           this command_group is independant from the first one
           and can be scheduled independently */

        // Enqueue a parallel kernel on a N*M 2D iteration space
        cgh.parallel_for<class init_b>({ N, M },
                            [=] (auto index) {
                              B[index] = index[0]*2014 + index[1]*42;
                            });
      });
    // Launch an asynchronous kernel to compute matrix addition c = a + b
    myQueue.submit([&](auto &cgh) {
        // In the kernel a and b are read, but c is written
        auto A = a.get_access<access::read>(cgh);
        auto B = b.get_access<access::read>(cgh);
        auto C = c.get_access<access::write>(cgh);
        // From these accessors, the SYCL runtime will ensure that when
        // this kernel is run, the kernels computing a and b completed

        // Enqueue a parallel kernel on a N*M 2D iteration space
        cgh.parallel_for<class matrix_add>({ N, M },
                                  [=] (auto index) {
                                    C[index] = A[index] + B[index];
                                  });
      });
    /* Request an access to read c from the host-side. The SYCL runtime
       ensures that c is ready when the accessor is returned */
    auto C = c.get_access<access::read, access::host_buffer>();
    std::cout << std::endl << "Result:" << std::endl;
    for(size_t i = 0; i < N; i++)
      for(size_t j = 0; j < M; j++)
        // Compare the result to the analytic value
        if (C[i][j] != i*(2 + 2014) + j*(1 + 42)) {
          std::cout << "Wrong value " << C[i][j] << " on element "
                    << i << ' ' << j << std::endl;
          exit(-1);
        }
  } /* End scope of myQueue, this wait for any remaining operations on the
       queue to complete */
  std::cout << "Good computation!" << std::endl;
  return 0;
}
```

Tasks with host dependencies and host accessors

```
#include <CL/sycl.hpp>
#include <iterator>
#include <numeric>
  cl::sycl::buffer<Type> b { N };
  {
    auto ab = b.get_access<cl::sycl::access::write>();
    // Initialize buffer b starting from the end with increasing
    // integer starting at 42
    std::iota(ab.rbegin(), ab.rend(), 42);
  }
  // A buffer of N Type to get the result
  cl::sycl::buffer<Type> c { N };
  // Launch a kernel to do the summation
  q.submit([&] (cl::sycl::handler &cgh) {
      // Get access to the data
      auto ab = b.get_access<cl::sycl::access::read>(cgh);
      auto ac = c.get_access<cl::sycl::access::write>(cgh);

      cgh.single_task<class sum>([=] {
        [...]
        });
    });
  // Wait for c to be available through this host accessor
  for(auto e : c.get_access<cl::sycl::access::read>())
    BOOST_CHECK(e == N + 42 - 1);
```

Outline

Pipes in OpenCL 2.*x*

- Simple FIFO objects
- Useful to create dataflow architectures between kernels without host
- Created on the host with some message size + object number
- `read()`/`write()` functions
- Same behaviour/guaranty as a memory buffer
  - For portability because no hardware FIFO mandatory in OpenCL 2.*x*
  - Can be implemented with a memory buffer
- Non blocking because no independent-forward-progress guaranty in execution model yet
  - No guaranty that a producer can run concurrently with a consumer
  - No guaranty between different work-items when blocking

Pipe on FPGA

- The actual motivation for pipes!
- External memory access main cause for power consumption...
- Real FIFO are easy to implement in hardware
  - Simple bus for 1 element FIFO
  - Latches or memory with more elements
- Very energy efficient
- Possible to have full dataflow applications without host control
- $\leadsto$ FPGA vendors provide OpenCL extensions for pipe with stronger guarantees
  - Xilinx evaluates pipe extensions for SYCL too

Producer/consumer with blocking pipe

```
#include <CL/sycl.hpp>
#include <iostream>
#include <iterator>

constexpr size_t N = 3;
using Vector = float[N];

int main() {
  Vector va = { 1, 2, 3 };
  Vector vb = { 5, 6, 8 };
  Vector vc;

  {
    // Create buffers from a & b vectors
    cl::sycl::buffer<float> ba { std::begin(va), std::end(va) };
    cl::sycl::buffer<float> bb { std::begin(vb), std::end(vb) };

    // A buffer of N float using the storage of vc
    cl::sycl::buffer<float> bc { vc, N };

    // A pipe of 2 float elements
    cl::sycl::pipe<float> p { 2 };

    // Create a queue to launch the kernels
    cl::sycl::queue q;

    // Launch the producer to stream A to the pipe
    q.submit([&](cl::sycl::handler &cgh) {
      // Get write access to the pipe
      auto kp = p.get_access<cl::sycl::access::write,
                             cl::sycl::access::blocking_pipe>(cgh);
      // Get read access to the data
      auto ka = ba.get_access<cl::sycl::access::read>(cgh);

      cgh.single_task<class producer>([=] {
          for (int i = 0; i != N; i++)
            kp << ka[i];
        });
      });

    // Launch the consumer that adds the pipe stream with B to C
    q.submit([&](cl::sycl::handler &cgh) {
      // Get read access to the pipe
      auto kp = p.get_access<cl::sycl::access::read,
                             cl::sycl::access::blocking_pipe>(cgh);

      // Get access to the input/output buffers
      auto kb = bb.get_access<cl::sycl::access::read>(cgh);
      auto kc = bc.get_access<cl::sycl::access::write>(cgh);

      cgh.single_task<class consumer>([=] {
          for (int i = 0; i != N; i++)
            kc[i] = kp.read() + kb[i];
        });
      });
  } /** End scope for the queue and the buffers:
        wait for completion q completion & bc copied back to v */

  std::cout << std::endl << "Result:" << std::endl;
  for(auto e : vc)
    std::cout << e << " ";
  std::cout << std::endl;
}
```

Non blocking pipe

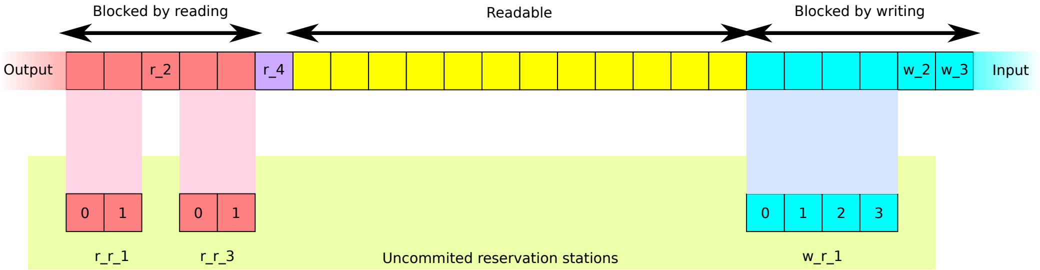
```
// Launch the producer to stream A to the pipe
q.submit([&](cl::sycl::handler &cgh) {
  // Get write access to the pipe
  auto p = P.get_access<cl::sycl::access::write>(cgh);
  // Get read access to the data
  auto ka = A.get_access<cl::sycl::access::read>(cgh);

  cgh.single_task<class producer>([=] {
      for (int i = 0; i != N; i++)
        // Try to write to the pipe up to success
        while (!(p << ka[i]))
          ;
    });
  });
```

Sequential pipe access

- FIFO: queue with serialized access
- How to implement simultaneous access by several work-item either at input or output?
- How to order access by work-items for deterministic execution?

⇝ Add concept of reservation station

Parallel pipe access with reservation



- Reservation station $\equiv$ array-view reserved in the pipe
- Allow ordered and parallel operation inside pipes
- Accessible up to commit operation
- Several reservation stations alive in parallel
- Can be mixed with on-going simple pipe access

Code example with reservation station

```
// Size of the buffers
constexpr size_t N = 200;
// Number of work-item per work-group
constexpr size_t WI = 20;
// The plumbing with some weird size prime to WI to exercise the system
cl::sycl::pipe<Type> pa { 2*WI + 7 };
// A buffer of N Type to get the result
cl::sycl::buffer<Type> c { N };
q.submit([&] (cl::sycl::handler &cgh) {
    // Get read access to the pipe
    auto apa = pa.get_access<cl::sycl::access::read,
                             cl::sycl::access::blocking_pipe>(cgh);
    // Get write access to the data
    auto ac = c.get_access<cl::sycl::access::write>(cgh);

    /* Create a kernel with WI work-items executed by work-groups of
       size WI, that is only 1 work-group of WI work-items */
    cgh.parallel_for_work_group<class consumer>(
      { WI, WI },
      [=] (auto group) {
        // Use a sequential loop in the work-group to stream chunks in order
        for (int start = 0; start != N; start += WI) {
          auto r = apa.reserve(WI);
          group.parallel_for_work_item([=] (cl::sycl::item<> i) {
              ac[start + i[0]] = r[i[0]];
            });
          // Here the reservation object goes out of scope: commit
        }
      });
  });
```

Co-scheduling

- Pipe connections make sense on FPGA for kernels... running at the same time ☺
  - Take advantage of direct 1-to-1 connections
  - On pageable FPGA with kernel reconfiguration, pipe-connected kernels need to be present at the same time
- SYCL provides a nice asynchronous execution model for distributed-memory system and accelerators
  - No provision for hard pipe-dependencies yet ☹
- $\exists$ co-scheduling: simultaneously scheduling of interdependent tasks
  - Useful on HPC machine to avoid resource waste
- $\rightsquigarrow$ SYCL scheduler needs to consider pipe-based relations too...

Outline

triSYCL

- Open Source implementation using templated C++1z classes
  - On-going implementation started at AMD and now lead by Xilinx
  - `https://github.com/amd/triSYCL`
  - 7 contributors
- Used by Khronos committee to define the standard
  - Languages are now too complex to be defined without implementation
- CPU-only implementation for now
  - Use OpenMP for computation + std :: thread for task graph
  - Quite useful for debugging
  - CPU emulation for free
- Looking for some interns ☺ to add outlining compiler to generate SPIR-V based on open source Clang/LLVM, etc.

Task execution (I)

```cpp
#include <memory>
#include <thread>
struct task : public std::enable_shared_from_this<task>,
              public detail::debug<task> {
  /** Add a new task to the task graph and
      schedule for execution */
  void schedule(std::function<void(void)> f) {
    /* To keep a copy of the task shared_ptr after
    the end of the command group, capture it by
    copy in the following lambda. This should be
    easier in C++17 with move semantics on capture
    */
    auto task = shared_from_this();
    auto execution = [=] {
      // Wait for the required tasks to be ready
      task->wait_for_producers();
      // Execute the kernel
      f();
      /* Release the buffers that have been
         written by this task */
      task->release_buffers();
      // Notify the waiting tasks that we are done
      task->notify_consumers();
      // Notify the queue we are done
      owner_queue->kernel_end();
    };
    /* Notify the queue that there is a kernel
       submitted to the queue. Do not do it in the
       task contructor so that we can deal with
       command group without kernel and if we put it
       inside the thread, the queue may have finished
       before the thread is scheduled */
    owner_queue->kernel_start();
    /* If in asynchronous execution mode, execute
       the functor in a new thread
       \todo it may be implementable with
       packaged_task that would deal with exceptions
       in kernels */
    std::thread thread(execution);
    /** Detach the thread since it will
        synchronize by its own means \todo This is an
        issue if there is an exception in the kernel */
    thread.detach();
  }
```

Task execution (II)

https://github.com/amd/triSYCL/blob/master/include/CL/sycl/command_group/detail/task.hpp#L60

Lazy implicit graph construction (I)

https://github.com/amd/triSYCL/blob/master/include/CL/sycl/buffer/detail/accessor.hpp#L112

```
/// Construct a host accessor from an existing buffer
accessor(detail::buffer<T, Dimensions> &target_buffer) :
  buf { &target_buffer }, array { target_buffer.access } {
  /* The host needs to wait for all the producers of the buffer to
     have finished */
  buf->wait();
}
/// Construct a device accessor from an existing buffer
accessor(detail::buffer<T, Dimensions> &target_buffer,
         handler &command_group_handler) :
  buf { &target_buffer }, array { target_buffer.access } {
  // Register the buffer to the task dependencies
  buffer_add_to_task(buf, &command_group_handler, is_write_access());
}
```

Lazy implicit graph construction (II)

Used from https://github.com/amd/triSYCL/blob/master/include/CL/sycl/command_group/detail/task.hpp#L60

```
#include <condition_variable>
  /// List of the buffers used by this task
  std::vector<detail::buffer_base *> buffers_in_use;

  /// The tasks producing the buffers used by this task
  std::vector<std::shared_ptr<detail::task>> producer_tasks;

  /// Store if the execution ended, to be notified by task_ready
  bool execution_ended = false;

  /// To signal when this task is ready
  std::condition_variable ready;

  /// To protect the access to the condition variable
```

Lazy implicit graph construction (III)

```
std::mutex ready_mutex;

/// Keep track of the queue used to submission to notify kernel completion
detail::queue *owner_queue;

/// Wait for the required producer tasks to be ready
void wait_for_producers() {
  for (auto &t : producer_tasks)
    t->wait();
  // We can let the producers rest in peace
  producer_tasks.clear();
}

/// Release the buffers that have  been used by this task
void release_buffers() {
  for (auto b: buffers_in_use)
    b->release();
```

Lazy implicit graph construction (IV)

```
  buffers_in_use.clear();
}

/// Notify the waiting tasks that we are done
void notify_consumers() {
  execution_ended = true;
  ready.notify_all();
}

/** Wait for this task to be ready

    This is to be called from another thread */
void wait() {
  std::unique_lock<std::mutex> ul { ready_mutex };
  ready.wait(ul, [&] { return execution_ended; });
}
```

Lazy implicit graph construction (V)

Other details in `https://github.com/amd/triSYCL/blob/master/include/CL/sycl/buffer/detail/buffer_base.hpp`

Outline

Future

- Not possible for a task to enqueue another task yet...
  - In OpenCL 2 it is possible but SYCL 1.2 targets OpenCL 1.2 hardware...
  - $\rightsquigarrow$ Add support for SYCL 2.*x*
- OpenCL 2 comes with shared virtual memory
  - SYCL 2.*x* will be able to allow this too
  - SYCL syntax without buffers
  - SVM may come with latency & power consumption...

Conclusion

- Task-oriented programming: common pattern for
  - Responsiveness even without hardware parallelism (concurrency)
  - Using hardware parallelism for better performance
  - C++11, TBB, pthreads (POSIX), OpenMP...
- Higher-level concept in SYCL C++ DSEL: task graph model
  - Schedule tasks according to dependencies
  - Deal with data motion across devices transparently
- SYCL buffers and accessors
  - Buffers: location-independent multi-dimensional arrays
  - Accessors: define how the storage is used (read/write/...) $\rightsquigarrow$ implicit task graph
- Model extensible to deal with hardware pipes on FPGA ($\rightsquigarrow$ SYCL 2.x?)
- SYCL DSEL task graph model is pretty generic and not only OpenCL-centric
  - Close to run-time such as StarPU and can deal with remote nodes, even with lower level API such as MPI, MCAPI...
  - Actually even not restricted to C++ either (SYPyL, SYJaL, SYJSCL, SYCaml...)