XILINX
ALL PROGRAMMABLE.

Experiments with triSYCL: poor (wo)man shared virtual memory

PPoPP 2016 SYCL workshop

Ronan Keryell
Xilinx Research Labs

2016/03/13

Outline

SYCL 1.2 ≡ pure C++14 DSEL

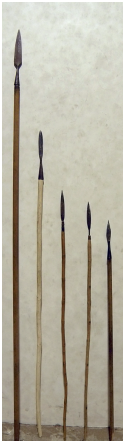
- Implement concepts useful for heterogeneous computing
- Asynchronous task graph
- Buffers to define location-independent storage
- Accessors to express usage for buffers and pipes: read/write/...
- Hierarchical parallelism
- Hierarchical storage
  - Rely on C++ allocator to specify storage
- Single source programming model
  - Take advantage of CUDA on steroids & 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 [ˈsi-kəl]

OpenCL SPIR



spear [ˈspir]

Future

- SYCL 2.*x* is coming...
  - Match OpenCL 2.*x* hardware features & C++17
  - SVM
  - kernel side enqueue
  - ...
- SYCL DSEL task graph model is pretty generic and not only OpenCL-centric
  - Close to run-time such as StarPU, Nanos++, OpenAMP... and can deal with remote nodes, even with lower level API such as MPI, MCAPI...
    - SYCL can target these runtimes!
  - Actually even not restricted to C++ either (SYPyCL, SYJaCL, SYJSCL, SYCaml...). SYFortranCL on top of Fortran 2008?

triSYCL

- Open Source implementation using templated C++1z classes
  - On-going implementation started at AMD and now led by Xilinx
  - `https://github.com/amd/triSYCL`
  - 8 contributors
- Used by Khronos committee to define the SYCL & OpenCL C++ standard
  - Languages are now too complex to be defined without implementing...
  - ∃ private Git repository for future Khronos & experimental Xilinx versions
- Pure C++ implementation & CPU-only implementation for now
  - Use OpenMP for computation + std::thread for task graph
  - Rely on STL & Boost for zen style
  - CPU emulation for free
    - Quite useful for debugging
  - More focused on correctness than performance for now (array bound check...)
- Looking for good interns ☺ to add outlining compiler to generate SPIR-V based on open source Clang/LLVM, etc.

Pipes on FPGA

- The actual motivation for pipes in OpenCL standard!
- 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 when more elements
  - Very energy efficient
  - High performance
- Possible to have full dataflow applications without host control
- ⇝ FPGA vendors provide OpenCL extensions for pipe with stronger guarantees
  - Blocking pipes ⇝ simpler applications
  - Static size possible ⇝ direct synthesis
  - Independent work-groups and kernels for producers/consumers connected with pipes

⇝ Xilinx evaluates pipe extensions for SYCL too

Producer/consumer with blocking pipe Xilinx extension

```
#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;
}
```

Motion detection on video in SYCL with blocking static pipes

```
auto window_name = "opencv_test";
cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);

cv::Mat rgb_data_in { NUMROWS, NUMCOLS, CV_8UC4 };
cv::Mat rgb_data_prev { NUMROWS, NUMCOLS, CV_8UC4 };
cv::Mat rgb_data_out { NUMROWS, NUMCOLS, CV_8UC4 };

cv::VideoCapture capture;
capture.open("./optical_flow_input.avi");

cv::Mat frame;
capture.read(frame);

cv::Mat small_frame;
cv::resize(frame, small_frame, rgb_data_in.size());

const int from_to[] = { 0, 0, 1, 1, 2, 2 };
cv::mixChannels(&small_frame, 1, &rgb_data_in, 1, from_to, 3);

cv::imshow(window_name, rgb_data_in);
cv::waitKey(30);

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

int framecnt = 0;
// Processing loop
while (capture.read(frame)) {
  cv::swap(rgb_data_in, rgb_data_prev);
  cv::resize(frame, small_frame, rgb_data_in.size());
  cv::mixChannels(&small_frame, 1, &rgb_data_in, 1, from_to, 3);
```

```
  {
    cl::sycl::buffer<int>
      buf_in { (int *) rgb_data_in.data, NUMROWS*NUMCOLS },
      buf_prev { (int *) rgb_data_prev.data, NUMROWS*NUMCOLS },
      buf_out { (int *) rgb_data_out.data, NUMROWS*NUMCOLS };

    // Send the images to the pipes
    read_data(q, buf_in, buf_prev);

     // Color conversion and sobel on the current image
    rgb_pad2ycbcr_in(q);
    sobel_filter_pass(q);

    // Color conversion and sobel on the previous image
    // \todo Unify rgb_pad2ycbcr_in and rgb_pad2ycbcr_prev
    rgb_pad2ycbcr_prev(q);
    // \todo Unify sobel_filter and sobel_filter_pass
    sobel_filter(q);

    // Compare 2 sobel outputs
    diff_image(q);
    combo_image(q, 0);

    // Color conversion and receive image from pipe
    ycbcr2rgb_pad(q);
    write_data(q, buf_out);
  }
  std::cout << "frame_" << framecnt++ << "_done\n";
  cv::imshow(window_name, rgb_data_out);
  cv::waitKey(30);
}
```

triSYCL C++ for FPGA

- Xilinx FPGA
  - Clocks
  - AXI ports
  - AXI stream ports
  - Interrupt ports
  - I/O devices (Ethernet, Interlaken, ADC, DAC, video...)
  - Reset
  - Dynamic Voltage and Frequency Scaling (DVFS)
  - Dynamic reconfiguration
  - Kernel scheduling
- Use native kernels to access specific I/O & IP
  - Single source C++ ⇝ hidden in "normal" class interface

- Add location/placement in device & sub-device selectors
- Use C++11 allocators to select memory type & location
- Use accessors to define read/write/bus (AXI4 master/slave/...)/pipe/linear/... data access
- Use SystemC-like data types for user-defined size & precision
- C++: normal API to control run-time & OS
- Tool metadata can be moved optionally from XML/TCL/JSON/... into C++ classes
  - Metaprogramming HLS ☺

Outline

Outline

On one side: C/C++ programming model

- Inherited from Von Neumann-Eckert-Mauchly machines...
- Single memory system with data and instructions accessed by addresses
- Data structures based on pointers (address abstraction)
- Typical computer job
  - Dereferencing pointers
  - Pointer chasing to walk data structures

On the other side: heterogeneous computing (I)

- Moving data is energy-hungry and slow...
  - A 32-bit integer computation is several orders of magnitude less costly than fetching the value from memory
- ⇝ Host computer connected to massively parallel specialized accelerators working on local memory
  - Too slow and inefficient to share main memory
- Simplified memory system on accelerator
  - Often no full-fledged virtual memory
  - Sometime different address size (64-bit vs 16-bit address bus)
- Complex explicitly managed memory hierarchy
  - Global, local, private address spaces in OpenCL
  - External DDR RAM, URAM, BRAM, registers in Xilinx FPGA
- No direct relation between address on host and accelerator
  - Pointer on 1 side completely unrelated on the other side ☹
- No direct sharing anyway... ☹

¿¿¿How to logically share complex data structures???

Thanks to Stefan Zellmann

- This talks is a side effect of `https://github.com/amd/triSYCL/issues/11`

Outline

Use real shared memory

- Shared virtual memory in OpenCL 2
  - Coarse grain buffer SVM guarantees that addresses inside buffers are the same on host and device
  - Fine grain SVM modes add finer interaction between host an accelerator
  - Coming soon in next SYCL version too
- Simple to use: almost no change in software ☺
- Requires some hardware support
  - But simple for coarse grain buffer SVM: adder on address bus for translation or MMU
- Cannot be possible in very efficient and heterogeneous architectures (different address sizes, not shared bus...)
- May be less efficient than a share-nothing architecture

Marshalling/unmarshalling data structure

- Serialize data structure
  - ASN.1, XML, JSON, YAML, ProtoBuf...
- Prettyprinter/parser on both host and accelerator side
  - Can allow data-compression & reorganization/defragmentation...
- Allows different memory organizations on host and device
  - Allows different address spaces with different address sizes
- Big cost at transfer time
  - Translation even for data that may not be used
- No execution cost once the translation is done

Pure software translation

- Off-line translation
  - Bulk transfer and walk data structure with in place translation
  - Big cost at transfer time
  - Not intrusive otherwise
- On-line translation
  - Rewrite code of each access on the accelerator or host
    - Add code around each pointer use to do address translation
  - Can allow efficient raw copy of data structures
  - No 1-time big translation cost
  - Small cost added to each access
  - Allows different address spaces with different address sizes
  - According to application, more interesting to do either on host or accelerator side
    - Translation on host may minimize storage if smaller address on accelerator
    - Accelerators are fast on parallel operations
    - Translation address can be efficient on GPU & FPGA: 1 addition

Outline

Running example: list-like data structure in plain C++

```
#include <iostream>

constexpr std::size_t N = 4;

template <typename T, std::size_t S>
struct simple_list {

  struct element {
    T value;
    element *next = nullptr;
  };

  element alloc[N];

  void walk(const element &e) const {
    std::cout << "Value = " << e.value
              << std::endl;
    if (e.next)
      walk(*e.next);
  }

};
```

```
int main(int argc, char *argv[]) {
  simple_list<int, N> a;

  // Initialize the data structure
  for (std::size_t i = 0; i != N; ++i) {
    a.alloc[i].value = i;
    if (i < N - 1)
      // Link current element to its successor
      a.alloc[i].next = &a.alloc[i + 1];
  }

  a.walk(a.alloc[0]);

  a.alloc[0].next = &a.alloc[2];
  a.alloc[2].value = 42;
  a.alloc[2].next = nullptr;

  a.walk(a.alloc[0]);

  return 0;
}
```

Running the example...

- Original data structure

@	value	next
0x7ffd1f4ece60	0	0x7ffd1f4ece70
0x7ffd1f4ece70	1	0x7ffd1f4ece80
0x7ffd1f4ece80	2	0x7ffd1f4ece90
0x7ffd1f4ece90	3	0

Value = 0
Value = 1
Value = 2
Value = 3

- Reorganized data structure

@	value	next
0x7ffd1f4ece60	0	0x7ffd1f4ece80
0x7ffd1f4ece70	1	0x7ffd1f4ece80
0x7ffd1f4ece80	42	0
0x7ffd1f4ece90	3	0

Value = 0
Value = 42

List-like data structure in SYCL (wrong version)

```
#include <iostream>
constexpr std::size_t N = 4;

template <typename T, std::size_t S>
struct simple_list {
  struct element {
    T value;
    element *next = nullptr;
  };

  element alloc[N];

  void walk(const element &e) const {
    std::cout << "Value = " << e.value
              << std::endl;
    if (e.next)
      walk(*e.next);
  }
};

int main(int argc, char *argv[]) {
  simple_list<int, N> a;

  // Initialize the data structure
  for (std::size_t i = 0; i != N; ++i) {
    a.alloc[i].value = i;
    if (i < N - 1)
      // Link current element to its successor
      a.alloc[i].next = &a.alloc[i + 1];
  }

  a.walk(a.alloc[0]);

  cl::sycl::buffer<simple_list<int, N>> d { &a, 1 };

  cl::sycl::queue {}.submit([&](cl::sycl::handler &cgh) {
      // request read-write access to our buffer
      auto da =
        d.get_access<cl::sycl::access::read_write>(cgh);

      // enqueue a single, simple task
      cgh.single_task<class update_list>([=] () {
          da->walk(da->alloc[0]);

          da->alloc[0].next = &da->alloc[2];
          da->alloc[2].value = 42;
          da->alloc[2].next = nullptr;

          da->walk(da->alloc[0]);
      });
    });

  a.walk(a.alloc[0]);

  return 0;
}
```

Running plain (wrong) SYCL example...

- Original data structure

@	value	next
0x7fffbfa1ce10	0	0x7fffbfa1ce20
0x7fffbfa1ce20	1	0x7fffbfa1ce30
0x7fffbfa1ce30	2	0x7fffbfa1ce40
0x7fffbfa1ce40	3	0

- On the device

@	value	next
0x7fffbfa1ce50	0	0x7fffbfa1ce20
0x7fffbfa1ce60	1	0x7fffbfa1ce30
0x7fffbfa1ce70	2	0x7fffbfa1ce40
0x7fffbfa1ce80	3	0

- Reorganized on device

@	value	next
0x7fffbfa1ce50	0	0x7fffbfa1ce70
0x7fffbfa1ce60	1	0x7fffbfa1ce30
0x7fffbfa1ce70	42	0
0x7fffbfa1ce80	3	0

- Back on the host

@	value	next
0x7fffbfa1ce10	0	0x7fffbfa1ce70
0x7fffbfa1ce20	1	0x7fffbfa1ce30
0x7fffbfa1ce30	42	0
0x7fffbfa1ce40	3	0

- ⚠ Crash forecast...
- Work if SVM or on host device

Fixing this in SYCL the C way

- Only use host addresses even on device
- Replace manually all the pointer operations by host→device translation + pointer operation (`*`, `->`, `[]`)
- Replace manually all the address operations by device→host translation + address operation (`&`, `= nullptr`)
- Frankenstein's intrusive coding style... ☹

Outline

Modern C++ support for dealing with addresses (I)

SYCL is pure C++!

- Operators can be overloaded in C++ ☺: `&`, `*`, `->`, `[]`
  - Use device→host or device←host translation in operations (`&`, `= nullptr`, `*`,...)
  - Unfortunately operator $*$ overloadable only for an object, not for a pointer type...
    - May require to use proxy object to represent device address...
- `nullptr_t nullptr` replaces old void $*$NULL
  - Specific type ⇝ possible overloading instead of dynamic test!
- `std::addressof` useful to return the *real* address
- Iterators are higher-level abstract pointers...
  - Could hide address translation
- auto keywords use type inference to avoid a lot of typing
- Allocator manages memory allocation and pointer definition
  - Require application to use pointer_traits

Modern C++ support for dealing with addresses (II)

- When you are completely lost... Static RTTI `typeid(some_thing).name()` give you the type name ☺
- Preprocessor macro `__SYCL_DEVICE_ONLY__` when compiling for device to specialize code on host/device

SYCL example with C++-style address translation (I)

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

constexpr std::size_t N = 4;

template <const std::intptr_t &Translation, typename T>
struct translated;

template <const std::intptr_t &Translation, typename T>
struct translated_address {
  using tt = translated<Translation, T>;
  // Keep the address from the internal address space
  tt *address;

  /// Convert a translated address into a normalized internal address
  static auto internalize(tt *external) {
    return reinterpret_cast<tt *>(reinterpret_cast<char *>(external)
                                  + Translation);
  }

  /// Return the address converted to the translated address space
  auto externalized() {
    return reinterpret_cast<tt *>(reinterpret_cast<char *>(address)
                                  - Translation);
  }
```

SYCL example with C++-style address translation (II)

```
/// Return the address converted to the translated address space
auto externalized() const {
  return reinterpret_cast<const tt *>(reinterpret_cast<const char *>(address)
                                      - Translation);
}

/// Construct a translated address from the address of a translated type
translated_address(tt *orig) : address { internalize(orig) } {}

// To have access to the internal address
auto value() const {
  return address;
}

/** The beauty of having nullptr instead of NULL

    This avoid testing for null in previous constructor */
translated_address(std::nullptr_t n) : address { nullptr } {}

tt & operator*() {
  return *externalized();
}

const tt & operator*() const {
  return *externalized();
```

SYCL example with C++-style address translation (III)

```
  }

  tt * operator->() {
    return externalized();
  }

  tt & operator[](std::size_t offset) {
    return externalized()[offset];
  }

  /** Implicit conversion operator to a reference on the address so it
      can behave like a tt address for other operations, such as += or
      differentes */
  operator tt *&() {
    return address;
  }

  /** Conversion to bool to test for nullptr */
  explicit operator bool() const {
    return address;
  }
};

/** A translated T that inherit from a T to behave like a T
 */
template <const std::intptr_t &Translation, typename T>
```

SYCL example with C++-style address translation (IV)

```
struct translated : T {

  // Provide the same constructors and destructors as a T
  using T::T;

  // Return a translated_address instead of a real address
  translated_address<Translation, T> operator &() {
    return this;
  }

};


template <typename T, std::size_t S>
struct simple_list {
  static std::intptr_t translation;

  struct element {
    T value;
    translated_address<translation, element> next = nullptr;

  };

  translated<translation, element> alloc[N];

  void walk(const element &e) const {
```

SYCL example with C++-style address translation (V)

```
    std::cout << "Value = " << e.value
              << std::endl;
    if (e.next)
      walk(*e.next);
  }
};

template <typename T, std::size_t S>
std::intptr_t simple_list<T, S>::translation = 0;


int main(int argc, char *argv[]) {
  simple_list<int, N> a;

  // Initialize the data structure
  for (std::size_t i = 0; i != N; ++i) {
    a.alloc[i].value = i;
    if (i < N - 1)
      // Link current element to its successor
      a.alloc[i].next = &a.alloc[i + 1];
  }

  a.walk(a.alloc[0]);

  auto a_base_address = reinterpret_cast<const char *>(a.alloc);
```

SYCL example with C++-style address translation (VI)

```
  cl::sycl::buffer<simple_list<int, N>> d { &a, 1 };

  cl::sycl::queue {}.submit([&](cl::sycl::handler &cgh) {
      // request access to our buffer
      auto da = d.get_access<cl::sycl::access::read_write>(cgh);

      // enqueue a single, simple task
      cgh.single_task<class update_list>([=] () {
          // Initialize the global translation parameter
          simple_list<int, N>::translation = a_base_address
            - reinterpret_cast<const char *>(da->alloc);
          da->walk(da->alloc[0]);

          da->alloc[0].next = &da->alloc[2];
          da->alloc[2].value = 42;
          da->alloc[2].next = nullptr;

          da->walk(da->alloc[0]);
      });
    });
  a.walk(a.alloc[0]);

  return 0;
}
```

Running SYCL example with C++ style translation...

- Original data structure

@	value	next	next.externalized()
0x7ffe2e262b70	0	0x7ffe2e262b80	0x7ffe2e262b80
0x7ffe2e262b80	1	0x7ffe2e262b90	0x7ffe2e262b90
0x7ffe2e262b90	2	0x7ffe2e262ba0	0x7ffe2e262ba0
0x7ffe2e262ba0	3	0	0

- On the device

@	value	next	next.externalized()
0x7ffe2e262bb0	0	0x7ffe2e262b80	0x7ffe2e262bc0
0x7ffe2e262bc0	1	0x7ffe2e262b90	0x7ffe2e262bd0
0x7ffe2e262bd0	2	0x7ffe2e262ba0	0x7ffe2e262be0
0x7ffe2e262be0	3	0	0x40

- Reorganized on device

@	value	next	next.externalized()
0x7ffe2e262bb0	0	0x7ffe2e262b90	0x7ffe2e262bd0
0x7ffe2e262bc0	1	0x7ffe2e262b90	0x7ffe2e262bd0
0x7ffe2e262bd0	42	0	0x40
0x7ffe2e262be0	3	0	0x40

- Back on the host

@	value	next	next.externalized()
0x7ffe2e262b70	0	0x7ffe2e262b90	0x7ffe2e262b90
0x7ffe2e262b80	1	0x7ffe2e262b90	0x7ffe2e262b90
0x7ffe2e262b90	42	0	0
0x7ffe2e262ba0	3	0	0

Outline

Writing these slides is painful... ☹

- But LaTeX beamer is a Turing-complete language... ☺
- Actually the previous slides were generated by running real examples

```
\begin{itemize}
\item Original data structure

  \input{/home/keryell/Xilinx/Projects/OpenCL/SYCL/triSYCL/tests/poor_woman_svm/simple_list_a_allocation.tex}
  \vavers
  \begin{minipage}{0.3\linewidth}
    Value = 0\\
    Value = 1\\
    Value = 2\\
    Value = 3
  \end{minipage}
\item Reorganized data structure

  \input{/home/keryell/Xilinx/Projects/OpenCL/SYCL/triSYCL/tests/poor_woman_svm/simple_list_a_allocation_reorganized.tex}
  \vavers
  \begin{minipage}{0.3\linewidth}
    Value = 0\\
    Value = 42
  \end{minipage}
\end{itemize}
```

- The code C++ code is instrumented to generate the slide source code

Special effects: the memory diagram LaTeX source code...

```
\begin{tabular}{r|c|c|}
  \cline{2-3}
  @ & \texttt{value} & \texttt{next} \\\cline{2-3}\hline
  \texttt{0x7ffd1f4ece60}\PlaceTextNode{0x7ffd1f4ece60}{} & \texttt{0} & \PlaceTextNode{1-0x7ffd1f4ece70}{}\texttt{0x7ffd1f4ece70} \\\cline{2-
  \texttt{0x7ffd1f4ece70}\PlaceTextNode{0x7ffd1f4ece70}{} & \texttt{1} & \PlaceTextNode{2-0x7ffd1f4ece80}{}\texttt{0x7ffd1f4ece80} \\\cline{2-
  \texttt{0x7ffd1f4ece80}\PlaceTextNode{0x7ffd1f4ece80}{} & \texttt{2} & \PlaceTextNode{3-0x7ffd1f4ece90}{}\texttt{0x7ffd1f4ece90} \\\cline{2-
  \texttt{0x7ffd1f4ece90}\PlaceTextNode{0x7ffd1f4ece90}{} & \texttt{3} & \PlaceTextNode{4-0}{}\texttt{0} \\\cline{2-3}
\end{tabular}
\begin{tikzpicture}[remember picture,overlay]
  \draw [overlay,->,very thick,red] (1-0x7ffd1f4ece70) to[bend right] (0x7ffd1f4ece70);
  \draw [overlay,->,very thick,red] (2-0x7ffd1f4ece80) to[bend right] (0x7ffd1f4ece80);
  \draw [overlay,->,very thick,red] (3-0x7ffd1f4ece90) to[bend right] (0x7ffd1f4ece90);
\end{tikzpicture}
```

```
\begin{tabular}{r|c|c|}
  \cline{2-3}
  @ & \texttt{value} & \texttt{next} \\\cline{2-3}\hline
  \texttt{0x7ffd1f4ece60}\PlaceTextNode{0x7ffd1f4ece60}{} & \texttt{0} & \PlaceTextNode{5-0x7ffd1f4ece80}{}\texttt{0x7ffd1f4ece80} \\\cline{2-
  \texttt{0x7ffd1f4ece70}\PlaceTextNode{0x7ffd1f4ece70}{} & \texttt{1} & \PlaceTextNode{6-0x7ffd1f4ece80}{}\texttt{0x7ffd1f4ece80} \\\cline{2-
  \texttt{0x7ffd1f4ece80}\PlaceTextNode{0x7ffd1f4ece80}{} & \texttt{42} & \PlaceTextNode{7-0}{}\texttt{0} \\\cline{2-3}
  \texttt{0x7ffd1f4ece90}\PlaceTextNode{0x7ffd1f4ece90}{} & \texttt{3} & \PlaceTextNode{8-0}{}\texttt{0} \\\cline{2-3}
\end{tabular}
\begin{tikzpicture}[remember picture,overlay]
  \draw [overlay,->,very thick,red] (5-0x7ffd1f4ece80) to[bend right] (0x7ffd1f4ece80);
  \draw [overlay,->,very thick,red] (6-0x7ffd1f4ece80) to[bend right] (0x7ffd1f4ece80);
\end{tikzpicture}
```

Solving the meta-problem: the full instrumented running example (I)

```
/* RUN: %{execute}%s

   Experiment with poor (wo)man SVM in plain C++
*/
#include <CL/sycl.hpp>
#include <fstream>
#include <iostream>
#include <set>
#include <sstream>

constexpr std::size_t N = 4;

template <typename T, std::size_t S>
struct simple_list {

  struct element {
    T value;
    element *next = nullptr;
  };

  element alloc[N];

  void walk(const element &e) const {
    std::cout << "Value = " << e.value
              << std::endl;
    if (e.next)
      walk(*e.next);
```

Solving the meta-problem: the full instrumented running example (II)

```
}

// Generate the name of a "from" node
static auto from_node_name(const void * address, int version) {
  std::ostringstream s;
  s << version << '-' << address;
  return s.str();
}

// Generate the name of a "to" node
static auto to_node_name(const void * address) {
  std::ostringstream s;
  s << address;
  return s.str();
}

void LaTeX_display(std::string file_name) const {
  // Number the drawing so we can have unique node labels
  static int drawing_number = 0;

  // Create a new output file, discarding old one if any
  std::ofstream output { file_name, std::ios_base::trunc };

  struct link {
    const void *from;
    int from_version;
    const void *to;
```

Solving the meta-problem: the full instrumented running example (III)

```
    link(const void *from, int from_version, const void *to)
      : from { from }, from_version { from_version }, to { to }
    {}
  };

  // Store the linking code to be output later
  std::vector<link> links;

  // To know if an address is about the local object or not
  std::set<const void *> local_addresses;

  output << R"(\begin{tabular}{r|c|c|}
 \cline{2-3}
 @ & \texttt{value} & \texttt{next} \\\cline{2-3}\hline)" << std::endl;
  for (const auto &e : alloc) {
    ++drawing_number;
    local_addresses.insert(std::addressof(e));

    // Display the address
    output << R"( \texttt{)" << std::addressof(e)
      // Address node landing pad
          << R"(}\PlaceTextNode{)" << to_node_name(std::addressof(e))
      // Display the value
          << R"(}{} & \texttt{)" << e.value
      // The node to point from
          << R"(} & \PlaceTextNode{)"
```

Solving the meta-problem: the full instrumented running example *(IV)*

```
            << from_node_name(e.next, drawing_number)
            << R"(}{}\texttt{})" << e.next
            << R"(} \\\cline{2-3})" << std::endl;
    // If the pointer is non null, add a link
    if (e.next)
      links.emplace_back(e.next, drawing_number, e.next);
  }
  output << R"(\end{tabular})" << std::endl;

  output << R"(\begin{tikzpicture}[remember picture,overlay])"
         << std::endl;
  for (const auto &link : links) {
    output << R"( \draw [overlay,->,very thick,)"
      // Red if link to a local address, blue otherwise
            << (local_addresses.count(link.to) ?
                "red" : "blue,loosely dotted")
            << "] ("
      // The node to point from
            << from_node_name(link.from, link.from_version)
            << ") to[bend right] ("
      // The node to point to
            << to_node_name(link.to)
            << ");"  << std::endl;
  }
  output << R"(\end{tikzpicture})" << std::endl;
}
```

Solving the meta-problem: the full instrumented running example (V)

```
};

int main(int argc, char *argv[]) {
  simple_list<int, N> a;

  // Initialize the data structure
  for (std::size_t i = 0; i != N; ++i) {
    a.alloc[i].value = i;
    if (i < N - 1)
      // Link current element to its successor
      a.alloc[i].next = &a.alloc[i + 1];
  }

  a.walk(a.alloc[0]);

  a.LaTeX_display("simple_list_a_allocation.tex");

  a.alloc[0].next = &a.alloc[2];
  a.alloc[2].value = 42;
  a.alloc[2].next = nullptr;

  a.walk(a.alloc[0]);

  a.LaTeX_display("simple_list_a_allocation_reorganized.tex");

  return 0;
}
```

Solving the meta-problem: the full instrumented running example *(VI)*

Outline

Conclusion

- SYCL C++ Khronos standard provides seamless single-source with OpenCL interoperability
  - Candidate for C++ SG14 standardization
  - SPIR-V gives portable execution model
  - SYCL ≡ pure C++ ⇝ integration with other C/C++ HPC frameworks: OpenCL, OpenMP, libraries (MPI, numerical Eigen/TensorFlow...), C++ DSeL (PGAS...)...
- ⇝ Simple generic method for providing poor (wo)man SVM
  - Pure C++ solution
  - Always use host address, even on device
  - Add 1 operation on device on all memory operations and address computation
  - Replace `T` with `translated<translation, T>`
  - Replace `T *` with `translated_address<translation, T>`
  - Generic programming: independent of `T` ⇝ provided as library possible
- triSYCL ≡ Open Source co-design tool for architectural & programming model exploration (FPGA, PiM/Near-Memory Computing, various computing models...)
  - Jump in the team!
- SYCL can be used to generate slides too!