14th International Workshop on OpenCL and SYCL

IWOCL 2026

Lightweight Tracing Interface for SYCL's USM Model Implemented in AdaptiveCpp

Jakob Niessner, Heidelberg University

Jakob Niessner, Aksel Alpay and Thomas Applencourt.
Heidelberg University and Argonne National Laboratory

May 6-8, 2026 | Heilbronn, Germany | iwocl.org

API Tracing Can Simplify Life a Lot

- Why do my operations not run concurrently?

API Tracing Can Simplify Life a Lot

- Why do my operations not run concurrently?
- Is there any non-deallocated memory in my application, if so where?

API Tracing Can Simplify Life a Lot

- Why do my operations not run concurrently?
- Is there any non-deallocated memory in my application, if so where?
- How often does my app use some API function such as `malloc_device`?

API Tracing Can Simplify Life a Lot

- Why do my operations not run concurrently?
- Is there any non-deallocated memory in my application, if so where?
- How often does my app use some API function such as `malloc_device`?
- What is a SYCL out-of-order queue mapped to in terms of in-order CUDA streams when using a CUDA backend?

API Tracing Can Simplify Life a Lot

- Why do my operations not run concurrently?
- Is there any non-deallocated memory in my application, if so where?
- How often does my app use some API function such as `malloc_device`?
- What is a SYCL out-of-order queue mapped to in terms of in-order CUDA streams when using a CUDA backend?

- Debugging

API Tracing Can Simplify Life a Lot

- Why do my operations not run concurrently?
- Is there any non-deallocated memory in my application, if so where?
- How often does my app use some API function such as `malloc_device`?
- What is a SYCL out-of-order queue mapped to in terms of in-order CUDA streams when using a CUDA backend?

- Debugging
- Performance tuning

API Tracing Can Simplify Life a Lot

- Why do my operations not run concurrently?
- Is there any non-deallocated memory in my application, if so where?
- How often does my app use some API function such as `malloc_device`?
- What is a SYCL out-of-order queue mapped to in terms of in-order CUDA streams when using a CUDA backend?

- Debugging
- Performance tuning
- Statistical code analysis

No Standardized Tooling Interface in SYCL

Most other programming models have tracing interface

No Standardized Tooling Interface in SYCL

Most other programming models have tracing interface

- OpenCL: OpenCL (Intercept) Layer

No Standardized Tooling Interface in SYCL

Most other programming models have tracing interface

- OpenCL: OpenCL (Intercept) Layer
- OpenMP: Tooling Interface

No Standardized Tooling Interface in SYCL

Most other programming models have tracing interface

- OpenCL: OpenCL (Intercept) Layer
- OpenMP: Tooling Interface
- CUDA: CUPTI

No Standardized Tooling Interface in SYCL

Most other programming models have tracing interface

- OpenCL: OpenCL (Intercept) Layer
- OpenMP: Tooling Interface
- CUDA: CUPTI
- OneAPI: XPTI

No Standardized Tooling Interface in SYCL

Most other programming models have tracing interface

- OpenCL: OpenCL (Intercept) Layer
- OpenMP: Tooling Interface
- CUDA: CUPTI
- OneAPI: XPTI

Shouldn't there be vendor non-specific standardized tooling layer for the SYCL API?

Intel's XPTI: Vendor Specific

- OneAPI: XPTI
  - Dispatcher/Subscriber
  - Builds DAG explicitly
  - Tracepoints
  - UID (Can be propagated all the way to driver level)
- Assumes a certain architecture for the SYCL runtime
- Hard to standardize via SYCL (also implement)

Basic Setup

- User-side

Basic Setup

- User-side

  Environment variable `SYCL_TOOL_LIBRARIES` with paths to tools

Basic Setup

- User-side

  Environment variable `SYCL_TOOL_LIBRARIES` with paths to tools
  Tools are loaded in the form of `*.so`-files (or eqv. in other OSs)

Developer Side

- Introduce trace points

Developer Side

- Introduce trace points
- Trace function is called whenever such a point is reached

Developer Side

- Introduce trace points
- Trace function is called whenever such a point is reached
- Expose functions without name mangling `extern "C"`

Developer Side

- Introduce trace points
- Trace function is called whenever such a point is reached
- Expose functions without name mangling `extern "C"`
- Keep tracer state in struct and type cast inside function

```
struct state_t{...};
...
Callback_function(void* usr_state, ...){
  state_t* state_ptr = (state_t*) usr_state;
  ...
}
```

Developer Side

```
struct state_t{...};
...
Callback_function(void* usr_state, ...){
  state_t* state_ptr = (state_t*) usr_state;
  ...
}

...

void init_register(){
 ...
}
```

Developer Side

```
...

void init_register(){
  auto* state = new state_t

  init_CallbackName(Callback);
  init_state(state);
  init_finalizer(Finalizer_function);

   ...
}
```

Two types of trace points

- Entry and exit points for API function calls

Two types of trace points

- Entry and exit points for API function calls
  - Trace point for entry and exit of the function

Two types of trace points

- Entry and exit points for API function calls
  - Trace point for entry and exit of the function
  - Example: `q.submit()` will trigger when submit function is entered and when exited

Two types of trace points

- Entry and exit points for API function calls
  - Trace point for entry and exit of the function
  - Example: `q.submit()` will trigger when submit function is entered and when exited
  - Does not see execution on device

Two types of trace points

- Entry and exit points for API function calls
  - Trace point for entry and exit of the function
  - Example: `q.submit()` will trigger when submit function is entered and when exited
  - Does not see execution on device
- Construction and destruction of SYCL runtime objects

Define start and end trace points

- Entry and exit points for API function calls
- Tracer takes some additional arguments to allow for more context

- `sycl::event`
  - `wait`
- `sycl::queue`
  - `submit`
  - `submit_secondary`
  - `wait`
- `sycl::malloc_*`

- `sycl::handler`
  - `memset`
  - `memcpy`
  - `copy`
  - `depends_on`
  - `parallel_for` / `parallel_for_work_group`
  - `single_task`
  - `fill`

Execute Callback Only for Contructor or Destructor of "Backend" Object

- Many SYCL objects, including `queue`, `event` are just shallow wrappers around native backend objects

Execute Callback Only for Contructor or Destructor of "Backend" Object

- Many SYCL objects, including `queue`, `event` are just shallow wrappers around native backend objects
- Copy/Move assignable/constructible

Execute Callback Only for Contructor or Destructor of "Backend" Object

- Many SYCL objects, including `queue`, `event` are just shallow wrappers around native backend objects
- Copy/Move assignable/constructible
- Are comparable, equality is an equivalency relation and equal objects have the same `std::hash`-value

Execute Callback Only for Contructor or Destructor of "Backend" Object

- Many SYCL objects, including `queue`, `event` are just shallow wrappers around native backend objects
- Copy/Move assignable/constructible
- Are comparable, equality is an equivalency relation and equal objects have the same `std::hash`-value
- Their destructor is called only when the last copy of an object is destroyed

Execute Callback Only for Contructor or Destructor of "Backend" Object

- Many SYCL objects, including `queue`, `event` are just shallow wrappers around native backend objects
- Copy/Move assignable/constructible
- Are comparable, equality is an equivalency relation and equal objects have the same `std::hash`-value
- Their destructor is called only when the last copy of an object is destroyed

Execute callback only when a unique new object is created!

Concatenation Rules

- Shorthand notation

Concatenation Rules

- Shorthand notation
  - `q.memcpy(...);` $\Longleftrightarrow$
    `q.submit([=](sycl::handler h){h.memcpy(...);});`

Concatenation Rules

- Shorthand notation
  - `q.memcpy(...);` $\Longleftrightarrow$
    `q.submit([=](sycl::handler h){h.memcpy(...);});`
  - `submit_start` $\Longrightarrow$ `memcpy_start` $\Longrightarrow$ `memcpy_end` $\Longrightarrow$
    `event_constructor` $\Longrightarrow$ `submit_end`

Concatenation Rules

- Shorthand notation
  - `q.memcpy(...);` $\Longleftrightarrow$
    `q.submit([=](sycl::handler h){h.memcpy(...);});`
  - `submit_start` $\Longrightarrow$ `memcpy_start` $\Longrightarrow$ `memcpy_end` $\Longrightarrow$
    `event_constructor` $\Longrightarrow$ `submit_end`
- Multiple Tools

Concatenation Rules

- Shorthand notation
  - `q.memcpy(...);` $\Longleftrightarrow$
    `q.submit([=](sycl::handler h){h.memcpy(...);});`
  - `submit_start` $\Longrightarrow$ `memcpy_start` $\Longrightarrow$ `memcpy_end` $\Longrightarrow$
    `event_constructor` $\Longrightarrow$ `submit_end`
- Multiple Tools
  - start-function and initializer in order in which they appear in `ENV`-variable

Concatenation Rules

- Shorthand notation
  - `q.memcpy(...);` $\Longleftrightarrow$
    `q.submit([=](sycl::handler h){h.memcpy(...);});`
  - `submit_start` $\Longrightarrow$ `memcpy_start` $\Longrightarrow$ `memcpy_end` $\Longrightarrow$
    `event_constructor` $\Longrightarrow$ `submit_end`
- Multiple Tools
  - start-function and initializer in order in which they appear in `ENV`-variable
  - end-function and finalizer in inverse order

Memory Leak Detector Can Be Easily Written

- `sycl::malloc_*_end` functions take pointer value as arguments

Memory Leak Detector Can Be Easily Written

- `sycl::malloc_*_end` functions take pointer value as arguments
- Combining with Boosts stack-trace utility, pointer allocation and file-lines can be associated

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

int main() {
  sycl::gpu_selector selector;
  sycl::queue q{selector};

  auto j = sycl::malloc_device<double>(100, q);
  auto k = sycl::malloc_shared<int>(100, q);
  auto l = sycl::malloc_host<float>(100, q);

  sycl::free(j, q);
  sycl::free(k, q);
}
```

Memory Leak Detector Can Be Easily Written

- `sycl::malloc_*_end` functions take pointer value as arguments
- Combining with Boosts stack-trace utility, pointer allocation and file-lines can be associated

```cpp
#include <sycl/sycl.hpp>

int main() {
  sycl::gpu_selector selector;
  sycl::queue q{selector};

  auto j = sycl::malloc_device<double>(100, q);
  auto k = sycl::malloc_shared<int>(100, q);
  auto l = sycl::malloc_host<float>(100, q);

  sycl::free(j, q);
  sycl::free(k, q);
}
```

```
Memory leak detector activated
memory allocated at:
main at
~/SYCL_tracing-examples/memory_leak.cc:9
but never freed!
```

Tally Tracer

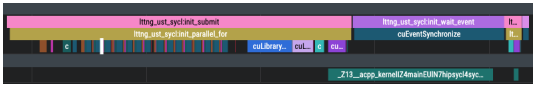
Name	Time	Time(%)	Calls	Average	Min	Max
submit	564.86ms	49.85%	18218	31.01us	6.38us	1.55ms
memcpy	457.85ms	40.41%	6074	75.38us	60.95us	1.54ms
parallel_for	54.79ms	4.84%	6072	9.02us	7.66us	173.39us
wait_event	53.72ms	4.74%	6072	8.85us	3.13us	14.89us
malloc_device	1.02ms	0.09%	3	338.51us	275.59us	402.26us
free	794.93us	0.07%	3	264.98us	223.55us	305.65us
wait_queue	14.85us	0.00%	2	7.42us	3.37us	11.48us
Total	1.13s	100.00%	36444			

Tally output of a Jacobi program from HeCBench on A100 GPU

- Measrues time spent and counts number of calls to an API function
- Example: Jacobi program from HecBench

Timeline visualization

```
sycl::queue Q{in_oder};
double *a = sycl::malloc_shared<double>(1, Q);
for (int i = 0; i < 2; i++) {
  Q.parallel_for(sycl::range<1>(1),
    [=](sycl::id<1> idx){
  a[0] = 42.0;
}).wait();
}
```

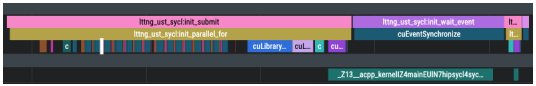


Timeline visualisation of the first call to a kernel with CUDA backend

- First `parallel_for` requires initialization step, second does not

Timeline visualization

```
sycl::queue Q{in_oder};
double *a = sycl::malloc_shared<double>(1, Q);
for (int i = 0; i < 2; i++) {
  Q.parallel_for(sycl::range<1>(1),
    [=](sycl::id<1> idx){
  a[0] = 42.0;
}).wait();
}
```

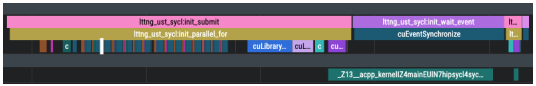


Timeline visualisation of the first call to a kernel with CUDA backend

- First `parallel_for` requires initialization step, second does not
- Possibility to target multiple layers at the same time

Timeline visualization

```
sycl::queue Q{in_oder};
double *a = sycl::malloc_shared<double>(1, Q);
for (int i = 0; i < 2; i++) {
  Q.parallel_for(sycl::range<1>(1),
    [=](sycl::id<1> idx){
  a[0] = 42.0;
}).wait();
}
```

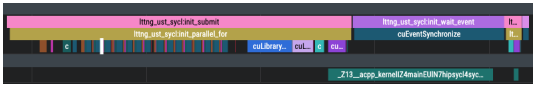


Timeline visualisation of the first call to a kernel with CUDA backend

- First `parallel_for` requires initialization step, second does not
- Possibility to target multiple layers at the same time
- Question: Are the moments when the SYCL runtime is not utilitzed

Timeline visualization

```
sycl::queue Q{in_oder};
double *a = sycl::malloc_shared<double>(1, Q);
for (int i = 0; i < 2; i++) {
  Q.parallel_for(sycl::range<1>(1),
    [=](sycl::id<1> idx){
  a[0] = 42.0;
}).wait();
}
```

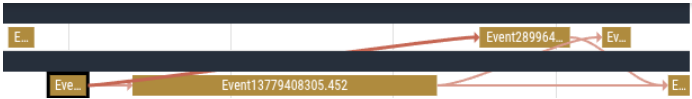


Timeline visualisation of the first call to a kernel with CUDA backend

- First `parallel_for` requires initialization step, second does not
- Possibility to target multiple layers at the same time
- Question: Are the moments when the SYCL runtime is not utilitzed
- How is `q.wait()` implemented in terms of CUDA API calls?

With the help of perfetto UI: Task graph visualizer

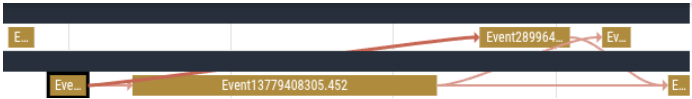
- Tracing the creation of (unique) events by hash value



Visualized task graph for a simple example program

With the help of perfetto UI: Task graph visualizer

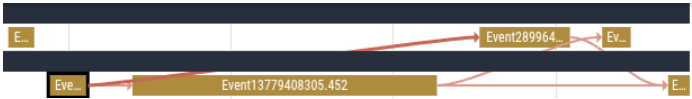
- Tracing the creation of (unique) events by hash value
- Possible to connect dependency of events (also for in-order queues)



Visualized task graph for a simple example program

With the help of perfetto UI: Task graph visualizer

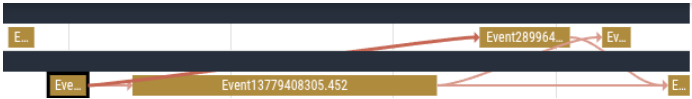
- Tracing the creation of (unique) events by hash value
- Possible to connect dependency of events (also for in-order queues)
- It is possible to draw the event graph



Visualized task graph for a simple example program

With the help of perfetto UI: Task graph visualizer

- Tracing the creation of (unique) events by hash value
- Possible to connect dependency of events (also for in-order queues)
- It is possible to draw the event graph

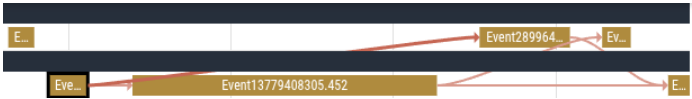


Visualized task graph for a simple example program

- Tracks represent queues

With the help of perfetto UI: Task graph visualizer

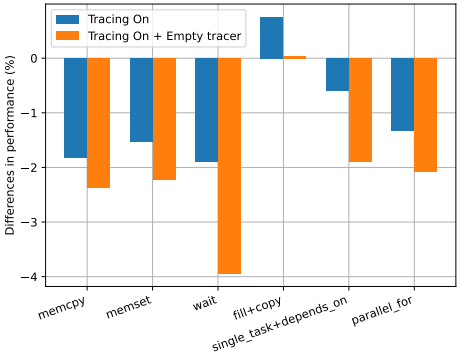
- Tracing the creation of (unique) events by hash value
- Possible to connect dependency of events (also for in-order queues)
- It is possible to draw the event graph



Visualized task graph for a simple example program

- Tracks represent queues
- With lower queue out-of-order) $\approx 30\%$ faster

Performance overhead negligible



- Performance comparison for trivial commands.
- API calls made per second in percent compared to AdaptiveCpp's develop branch
- Blue only tracing interface
- Orange: empty callbacks hooked in
- Tracing interface consisently introduces less 2% overhead

Contributions

- Defined a standard compliant lightweight tracing interface for SYCL's (a bit more than) USM model

Contributions

- Defined a standard compliant lightweight tracing interface for SYCL's (a bit more than) USM model
- Implemented it for AdaptiveCpp as a proof of concept

Contributions

- Defined a standard compliant lightweight tracing interface for SYCL's (a bit more than) USM model
- Implemented it for AdaptiveCpp as a proof of concept
- Showed that it brings minimal performance overhead

Contributions

- Defined a standard compliant lightweight tracing interface for SYCL's (a bit more than) USM model
- Implemented it for AdaptiveCpp as a proof of concept
- Showed that it brings minimal performance overhead
- Implemented some useful tools as a p.o.c.

Future Work

- Extent this interface to the buffer-accessor model

Future Work

- Extent this interface to the buffer-accessor model
- Implement more tools

Future Work

- Extent this interface to the buffer-accessor model
- Implement more tools
- Start a discussion on whether and how this might become part of the standard

Acknowledgements

Thomas Applencourt

Aksel Alpay

Argonne NATIONAL LABORATORY

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386