Accessors

—

a C++ standard library class to qualify data accesses

Ronan Keryell (Xilinx) Joël Falcou (NumScale)

September 30, 2016

Document	P0367R0
Date	2016-05-29
Project	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience	SG14, SG1, LEWG
Authors	Ronan Keryell (Xilinx), Joël Falcou (NumScale)
E-mails	`ronan.keryell at xilinx dot com` `joel.falcou at numscale dot com`
Reply to	`ronan.keryell at xilinx dot com`

Abstract

Accessing data is the most important aspect when it comes to high-performance computing or power efficiency in embedded computing. Furthermore, generalizing C++ to targets such as GPU or FPGA requires even finer control in the programmer hands.

We propose to abstract data accesses through an *accessor* class to give control to the programmer on how fine grain access is done, such as caching, memory burst or remote access in heterogeneous computing.

The `std::accessor<>` class is a proxy wrapper that behaves like the wrapped object but adds access properties to it or change the access behaviour.

For example if you have a slow I/O or a memory access (a special case of slow I/O nowadays...) but you know that pretty often the result is 42 for obvious reasons, you may rewrite your code

```
auto result = f(some_io());
```

to

```
auto result = f(make_accessor<likely> { some_io(), 42 });
```

and the compiler can decide for example to clone the execution of `f` to compute ahead `f(42)` or even to `constexpr`-evaluate it and the result is only committed when `some_io()` comes back and the value is verified as predicted. If not, the normal evaluation of `f()` goes on.

Contents

1 Motivation

Demand for high-performance and power efficiency makes architectural considerations more and more important when programming, specially with the generalization of distributed computing and heterogeneous computing involving accelerators, DSP, GPU, FPGA, network accelerators, etc.

Unfortunately, as for a sequential program running on a CPU, there is no performance portability when it is about reaching the maximum performance and power efficiency on a given architecture. Some execution parameters may have to be tweaked and/or the architecture of the software has to be deeply changed accordingly. Since there are more parameters under control in an heterogeneous platform compared to a CPU, the exploration space is quite wider. Dealing with this in an automatic way is an intractable issue in the general case but at least we should have some ways to express some of these details at the C++ level to reach maximum performance and power efficiency.

For example, currently there is no way to specify how the data are accessed and if we consider that now most of the energy consumption is spent in data transfer, specially with external memory, this is something to address.

Keeping data on a first-level cache in CPU is crucial and it is important to express which data will benefit or not from being in the cache. Since cache memories are very small and expensive, specifying that some data do not take advantage of the cache leaves more room for data in the critical path.

In the following we develop the concept of *accessor* represented as a plain C++ class to express how data are accessed in a C++ program.

An accessor is a proxy object that behaves like the object it represents but with some ways to change the behaviour when read or written.

Most of the behaviour could be done by language extensions or `#pragma`, but the advantage of having it as a class is that it can often be implemented in user-mode C++ for simplicity, portability or debug, and also implemented by a compiler in a target-specific optimized way on some architectures.

Having a plain object to control accesses provides handy RAII framework to hide actions in the accessor constructor and destructor, such as setting up the communication framework or switching on and off the power of the system that gives access to the object.

Having access properties encoded in the type itself allows propagation though generic templated function calls or lambda captures and allows code specialization with metaprogramming according to some access properties.

2 Related work

In current C++ standard and proposals or other libraries, some architectural aspects can already or will be addressed:

`thread` can be used to execute some code in parallel on multiple execution units;

`allocator` controls the way allocation happens and how pointers behave, and thus hide some hardware detail for data access;

constructors and destructors can hide some architectural details and semantics;

operator overloading is useful to hide some hardware operations

- `auto` **operator overloading** would make cleaner implementation of proxy to some hardware details and cleaning up expression templates;
- **operator dot overloading** [SR16] allows changing the behaviour and some extensions to generate function objects [GK15] allow interesting use case to change the object behaviour;

fixed-point type proposals are useful to have better performance on DSP and FPGA;

the concept of view has some similarities but is more focused on some kind of objects, such as `array_view` for arrays, `string_view` for strings, or `span` to view a sequence as a range:

- **array_ref** [ELT+16] is a proxy object to view some array-like objects in various ways;

SYCL is a C++ OpenCL standard from Khronos to execute some functors on some accelerators with possibly different address spaces, with the concept of accessors representing remote access to multidimensional arrays [WRRR16];

C++AMP and hcc [SSC+15] are C++ extension to execute some functors on some accelerators with possibly different address spaces, with the concept of `array_view` representing remote access to multidimensional arrays;

Boost.SIMD [SIM16] provides non-temporal iterators and ranges to work on vectors;

ISO/IEC TR 18037 in the C world, "Information Technology — Programming languages – C – Extensions to support embedded processors" (ISO/IEC JTC1 SC22 WG14 N1169, 2006-04-04 http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1169.pdf) introduces concepts such as the *named address spaces* to represent special kinds of memories, that went recycled into CUDA and OpenCL address spaces, or named registers and special I/O.

The concept of accessor discussed here is more focused on the action of reading or writing than on the kind of object.

3 Accessor

An `std::accessor<T, access_kind...>` is a variadic templated class that act as a proxy object to qualify how an original `T` object is accessed.

Since there are a lot of different ways to access an object, we prefer to have the accessor to be parametrized by the kind of accessor instead of having completely different explicit accessor classes:

```
auto io = get_some_memory_mapped_io();
std::accessor<write> use_it { io };
// Now we are sure this is a write-only usage
some_random_code(use_it);
```

with the type of the data accessed actually inferred from the constructor.

Accessors actually compose to define more complex access modes from basic ones, such as:

```
unsigned char buffer[N];
// Combine a write-only accessor with a modulo accessor to have a
// write-only circular buffer
std::accessor<std::accessor<modulo>, write> wcb { buffer };
generate_some_data(wcb);
```

The order of composition matters to match the requested semantics.

Of course the syntax becomes cumbersome and this is why the accessor is actually a variadic templated class so the previous example can be simplified as

```
unsigned char buffer[N];
// Combine a write-only accessor with a modulo accessor to have a
// write-only circular buffer
std::accessor<write, modulo> wcb { buffer };
generate_some_data(wcb);
```

with the accessor types combined from left to write in the order mathematical function composition: *make a write accessor on top of (from) a modulo accessor*. If some constructor arguments are required by the accessor there are matched in the reverse order.

A variadic templated factory method is also provided and usable as:

```
double a[10000];
auto p = std::make_accessor<prefetch<16>> { a };
// Use p as a but prefetch from memory 16 elements ahead
f(p);
```

3.1 Non unified memory

The current C++ memory model assumes more or less that the memory address space is uniform. Unfortunately, for HPC class machines from the Top 500 or for embedded systems, this simple addressing scheme does not hold and there are for example some private memory attached to each processor or device that cannot be addressed directly by another processor or device.

The motivation for this `accessor` proposal actually comes from OpenCL SYCL C++ where some functors are remotely executed but need to interact with a global memory. This is also the concept behind PGAS (Partitioned Global Address Space) languages or libraries, such as Coarray C++ or UPC++, where the memory is physically distributed but can be remotely accessed with some explicit language or library support instead of transparent virtual shared memory.

The use of a (SYCL) accessor can be seen on this SYCL C++ example:

```
// Create a 1D buffer of N double
cl::sycl::buffer<double, 1> b { N };
// [...]
// Launch on default accelerator:
cl::sycl::queue {}.submit([&](handler &cgh) {
  /* Get an accessor to write the data remotely from inside lambda
     which is off-loaded to the accelerator */
  auto acc = b.get_access<access::write>(cgh);
  // A DIY parallel equivalent of std::iota as a kernel named "init"
  cgh.parallel_for<class init>(N, [=] (auto index) {
                                  acc[index] = index;
                                 });
  });
```

could be generalized in a framework returning a remote accessor to allow remote access as:

```
std::array<double, N> local;
accessor<remote, write, std::array<double, N>> remote =
  get_some_runtime_remote(local);

std::copy(std::begin(local), std::end(local), std::begin(remote));
```

Typical environments usable with this could be SHMEM, Portal4, SYCL, Coarray C++, RDMA, iWARP, MPI...

Actually the real power from remote accessors would come from the combination with the `array_ref` [ELT+16] to have a simple implementation of the higher-level accessors found in SYCL or Coarray C++, or implement some for the other environments.

3.2 Read/write qualifiers

In C++ there is the `const` qualifier to specify read-only but there is no way to specify other access modes that normally do not change the semantics but are useful to increase performance or lower power consumption.

Here is a list of accessor types:

`write` to access to a write-only data. This avoids for example to load or prefetch the cache from memory before writing;

`read` to prevent writing back data;

`read_write` normal access by symmetry;

`discard_read` to read data if they are written first and do not read the initial value at accessor creation. A typical use case is a data locally generated that can be read back locally;

`discard_write` to write data to be locally read but won't be written back at accessor destruction;

`noaccess` by symmetry, typically to detect inappropriate behaviours in a program.

3.3 Non temporal access

A `non_temporal` accessor allows to access data that won't be accessed again and typically will not for example use a cache. This diminish cache transactions and leave the cache for some more useful usage.

For example when generating a huge amount of data to an array, everything else would be evicted from the cache without any chance to read the array back from the cache anyway.

A use case:

```
// An array of 1 TB
double a[2<<37];
auto ac = make_accessor<non_temporal,write>(a);
// Initialize a starting from the end with increasing
// integer starting at 42
std::iota(std::par_vec, ac.rbegin(), ac.rend(), 42);
```

3.4 Aliasing

Aliasing between different data structures may prevent the compiler from doing aggressive optimizations. Some proposals exist using generalize attributes [FTC+14] but it can also be done with accessors such as:

```
double a[N], b[N];
// Dummy class declarations to be used as alias set tags
class ta;
class tb;

auto ca = make_accessor<alias_tag<ta>>(build_complex_data_structure(a));
auto cb = make_accessor<alias_tag<tb>>(build_complex_data_structure(b));
auto oa = make_accessor<alias_tag<ta>>(other_complex_data_structure(a));
auto ob = make_accessor<alias_tag<tb>>(other_complex_data_structure(b));
// No aliasing
correlation(ca, cb);
correlation(oa, ob);
// Aliasing
correlation(ca, oa);
correlation(cb, ob);
```

Inside `correlation()` the compiler can use the aliasing information to optimize more or less the code, as with generalized attributes from [FTC+14], but with a type as a tag.

But since it is an accessor class, the programmer can query the accessor aliasing status with a type trait (§ 4) to know if there are aliasing between accessors and if so statically dispatch completely different algorithms.

3.5 Sequential access

In some case the programmer knows that accesses to an array are strictly sequential but the compiler cannot prove it. By explicitly specifying it, a compiler can generate vector memory access or do some parallel loop nest pipelining where the memory access is completely replaced for example by a hardware FIFO between execution units, specially in the case of low level targets (FPGA).

In the program

```
{
  std::accessor<accessor::sequential,
                accessor::discard_read,
                accessor::discard_write> a { some_array };
```

```
  for (int i; i = 0; i != N; ++i)
    a[i] = i;
  for (int i; i = 0; i != N; ++i)
    b[i] = a[i]*2;
}
```

the compiler could decide to fuse the 2 loops or to generate 2 Kahn's processes, 1 producer and 1 consumer, with only an efficient hardware FIFO in between and eluding the memory transfer on `a[i]`.

3.6 Prefetching

Latency is often a performance killer and if we know in advance that we will read some array elements, we could prefetch it. Note that prefetching is actually independent from caching, so it can be combined with non temporal access.

```
int a[N];
// Instruct the memory prefetcher to look 16 elements ahead
std::accessor<prefetch<16>> a_p { a };
std::fill(std::begin(a_p), std::end(a_p), 0);
```

3.7 Burst mode

Most of the memory interfaces work better when memory transfers are made with a coarse granularity. It can be specified with this accessor.

Note that since burst mode uses bulk transfer mode, it is not interesting for example when transferring small data randomly placed. In this case a burst-size of 1 with a non temporal accessor can be used.

```
int a[N*20];
// Instruct the memory prefetcher to use a burst mode of 20 elements
std::accessor<burst<20>> a_p { a };

// To generate random integers between 0 and N - 1
std::default_random_engine r;
std::uniform_int_distribution<int> d { 0, N - 1 };
for (int i = 0; i != N; ++i) {
  // Randomly write blocks of 20 elements
  p = std::begin(a_p) + 20*d(r);
  std::fill(p, p + 20, 0);
}
```

3.8 Pipelined access

Software loop pipelining is a classical loop transformation to reduce intra-iteration dependency which has tremendous effect on low-end processors (micro-controller with no cache) or specialized architectures (FPGA), by rewriting

```
for (int i = 0; i < N; ++i)
  a[i] = f(b[i]);
```

(assuming $N \geq 2$) to:

```
// Prelude
auto r = b[0];
auto w = f(r);
r = b[1];
```

```
// Pipelined loop
for (int i = 0; i < N - 2; ++i) {
  a[i] = w;
  w = f(r); // For i + 1
  r = b[i + 2];
}
// Postlude
a[N - 2] = w;
a[N - 1] = f(r);
```

A compiler is expected to do this kind of transformations automatically but some times cannot figure out automatically if it is legal or what is the actual benefit, for example when iterating on some complex iterators. By using an `accessor<pipelinable>` the compiler can generate a pipelined loop:

```
auto f = [](auto input, auto output) {
  std::transform(std::cbegin(input), std::cend(input),
                 std::begin(output), func);
};

int in[N], out[N];

// Call a pipelined implementation of f without requiring
// interprocedural analysis of func
f(std::make_accessor<pipelinable> { in },
  std::make_accessor<pipelinable> { out });
```

3.9 DMA

To take advantage from DMA to transfer data, the transfer can be encoded as a DMA operation with a DMA accessor.

```
int far_far_away[N];
{
  int near_and_fast[N];
  auto p = make_accessor<dma> { far_far_away, near_and_fast };
  std::generate(p, p + 20, 0);
}
```

The read/write accessor mode can be used to remove useless copy in the constructor or the destructor.

More complex data transfers may be defined using static DMA descriptors (can be synthesized in hardware on FPGA) or dynamic DMA descriptors given in the accessor constructor.

3.10 Bus type

Some systems allow different types of buses an a variable may be addressed through these different buses with an accessor specifying a bus identifier. The concept is standard but the bus identifiers, besides the default one, are implementation specific:

```
char a;
float b;
std::accessor<bus<axi4>> a_a { a };
std::accessor<bus<axi4>> b_a { b };
std::accessor<bus<main> a_m { a };
std::accessor<bus<main> b_m { b };
// Use different buses in parallel to improve bandwidth
auto sum = a_a + b_m;
```

```
auto prod = a_m + b_a;
```

The synchronization constraints between the various buses are implementation dependent.

3.11 Access width

Embedded systems allow to specify the size of the data-packets transferred on a bus. This can be specified with:

```
double d;
// Transfer 8-bit at a time
std::accessor<bit_width<8>> d_b { d };
auto a = d_b;
```

3.12 Address mode

Some architectures allow different addressing modes with different trade-off, such as PC-relative, based on a base pointer, near to some page, etc. To compile efficiently with this mode, a specific accessor is provided.

The available modes are implementation dependent, besides the `std::accessor<address_mode<normal>>`.

3.13 Translation

In embedded systems, it is common to have some level of shared memory but mapped physically at different addresses. If there is no virtual memory in use in some part of a system, the address in the different point-of-views appear as translated by an offset.

```
unsigned char frame_buffer[N];
size_t offset = &display - frame_buffer;
// The screen memory on the display controller
std::accessor<translate> b { frame_buffer, offset };
```

3.14 Modulo addressing

Implementing some circular buffers may require some kind of modulo addressing and some processors have this addressing mode. Since it is impossible to detect automatically this feature in the compiler in the general case, it should be expressed with an accessor:

```
unsigned char buffer[N];
// b is a kind of infinite array, but with only buffer storage repeated
std::accessor<modulo> b { buffer };
```

A specialized version of [DO16] could use this accessor.

3.15 Address bit setting

Some architectures encode in the address bus some semantics which is not used as the part of the address itself, such as supervisor mode, non executable mode, etc.

A bit-setting accessor allows to change the address bit accordingly during read or write operations:

```
int a[N];
a[0] = 2;
// On the target architecture, the address space is duplicated on the
// half upper 32KB space for a cacheless access
```

```
std::accessor<bit_set<or<(1 << 15)>>> cacheless_access { a };
// as a[123] but do not use the cache
cacheless_access[123] = 4;
```

3.16 Transactional memory

A transactional-memory accessor start a transaction at the accessor creation up to its destruction, with some transaction behaviour for the threads using this accessor. In case of write data-race, only one of the conflicting threads are not rolled back up to the construction of the accessor.

For example the 2 following functions

```
int shared_data[N];

void f() {
  std::accessor<transaction> a { shared_data };
  foo(a);
}

void g() {
  std::accessor<transaction> b { shared_data };
  bar(b);
}
```

may be executed from 2 different threads and will use a transactional memory behaviour if available, or otherwise the implementation will fallback on a lock-based solution.

3.17 Prediction

Sometime the programmer knows some probabilities about the distribution of values and this can be useful for probabilistic optimization. For example in

```
int v = some_io();
// Execute f() ahead knowing that most of the time v is 0
std::accessor<likely> prediction { v, 0 };
auto result = f(prediction);
```

the compiler can decide for example to clone the execution of `f` with some predication to remove side effects while doing a speculative execution. The actual `result` will be really committed only when `v` come out from the network and is compared to the predicted value.

We can have a PGO (Profile-Guided Optimization) version of it to instruct the compiler to instrument the code to do some statistical analysis of the most common value:

```
// Execute f() ahead after some PGO analysis to figure out common values
auto result = f(make_accessor<pgo_likely> { some_io() });
```

We can allow several likely values too.

3.18 Generic proxy

A proxy accessor delegates all the read and write operations to some user-provided functors.

```
void instrument(int v[]) {
  std::accessor<proxy> p { v, read_functor, write_functor };
  // Call f on p instead of v to intercept the read and write
  f(p);
}
```

It is useful for example for:

- virtualizing some non existant memory or hardware;
- implementing transactional memory in user mode;
- testing with some non existing software part by interacting with a mock-up hidden behind an accessor;
- override the memory operation to do fault injection for fault-tolerance evaluation;
- security testing with fuzzing of inputs.

4 Type traits

Having classes to qualify accesses makes possible some metaprogramming according to these type properties.

For this there are some type traits to introspect accessors at compile time, in the form of `std::is_accessor<`*`property_list`*`>(acc)` or `std::get_accessor<`*`property_list`*`>(acc)`.

For example it is possible to test if 2 accessor types are aliasing or not:

```
auto correlation = [](auto data1, auto data2) {
  if constexpr (std::is_accessor_v<aliasing_with>(data1, data2))
    slow_conservative_correlation(data1, data2);
  else
    crazy_aggressive_correlation(data1, data2);
};
```

to go back to the example from § 3.4.

5 Implicit accessor

Having to use explicit `accessor` objects may be painfully intrusive and it would be nice to have something lighter. Of course, since omnipotent abstract interpretation of a program is impossible, some kind of program transformation is required by the programmer to express properties of memory access.

We can use language extensions, `#pragma` or generalized attributes. Language extensions are bad for portability and acceptance. `#pragma` do not compose well with meta-programming. So we focus here on decorating objects with accessors.

```
// An array of 1 TB
double a[2<<37] [[std::accessor<non_temporal,sequential>]];

// An implicit accessor is wrapped around all uses of a
std::iota(a.begin(), a.end(), 0);
```

But also on any scope, such as class, block, namespace... as for example to add a behaviour of a transactional memory on all a class:

```
template <typename T>
class message_queue [[std::accessor<transaction>]] {
  T read() {...}

  void write(T &&t) {...}
}
```

6 Implementation ideas

The implementation basics for the proxy objects are:

- for fundamental types, have an implicit conversion operator to the reference to the fundamental types so the accessor can behave like the fundamental type;
- for object types, the proxy would publicly inherit from the type to forward all member access to it;
- the concept of accessor can just inherits from a reference accessor class implementing the proxy behaviour according to the basic type (fundamental, array, class, pointer);
- each kind of accessor inherits from another accessor class and adds its own properties to it as member types and optional member variables and methods, so at the end a full accessor is an aggregation as an inheritance list;
- some accessors such as the generic proxy accessor may require real compiler support.

7 Issues

Having some objects appearing at other addresses may put some restriction on the type (such as "trivially copyable", "standard layout"...).

8 Conclusion

Since accessing data is a real issue today for performance and power efficiency reasons, we introduce the concept of accessor to give the programmer some ways to optimize data accesses or extend data accesses beyond Von Neuman's architecture, the natural scope of C/C++, involving distributed memory and heterogeneous architectures.

We propose to control various aspects such as simple as read/write access control down to hardware bus selection, cache control, pipelining, etc.

Instead of extending the language, we propose to encapsulate accesses in normal STL classes, the `std::accessor`, to wrap up objects and add properties to their accesses. This class is templated to encode various properties that compose nicely.

Having this information available in the type system allows introspection and metaprogramming at compile time with specialization according to the types of possible data accesses.

Combined with concepts such as executors and ranges, it allows building very-high level parallel distributed applications with very extreme low-level bare-metal optimizations.

Acknowledgements

We want to thank all the people from the Khronos OpenCL SYCL committee for the fruitful discussions leading to this generalization of the concept of accessor.

Acknowledgments go to Xilinx for supporting this work and also to colleagues for their fruitful discussions on advanced C++ for FPGA: specially Ralph Wittig, Jeff Fifield and Sam Bayliss.

Thanks to Lee Howes for his PhD [How10] research on some concepts close to accessors and for the discussions on SYCL accessors.

Thanks to Michael Wong for helping bootstrapping this proposal in the C++ SG14 committee.

The PIPS team from MINES ParisTech and the Par4All team at SILKAN are thanked for the feedback on compilers using polyhedral techniques for automatic parallelization got heterogeneous computing and how we could have "array regions" (polyhedral approximations of accesses) in accessors, even it did not get trough this proposal yet.

Thanks to Albert Cohen for the discussion on how we could have the PENCIL IR [BCG+13] into C++, even it did not get trough this proposal yet.

References

[BCG+13] Riyadh Baghdadi, Albert Cohen, Serge Guelton, Sven Verdoolaege, Jun Inoue, Tobias Grosser, Georgia Kouveli, Alexey Kravets, Anton Lokhmotov, Cedric Nugteren, Fraser Waters, and Alastair F. Donaldson. PENCIL: Towards a platform-neutral compute intermediate language for dsls. Technical report, February 2013. http://arxiv.org/abs/1302.558.

[DO16] Guy Davidson and Arthur O'Dwyer. A proposal to add a ring span to the standard library. Technical Report P0059R1, February 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0059r1.pdf.

[ELT+16] H. Carter Edwards, Bryce Lelbach, Christian Trott, Juan Alday, Jesse Perla, Mauro Bianco, Robin Maffeo, and Ben Sander. Polymorphic multidimensional array reference. Technical Report P0009R1, February 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0009r1.html.

[FTC+14] Hal Finkel, Hubert Tong, Chandler Carruth, Clark Nelson, Daveed Vandevoode, and Michael Wong. Towards restrict-like aliasing semantics for c++. Technical Report N3988, May 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3988.pdf.

[GK15] Mathias Gaunard and Dietmar Kühl. Function object-based overloading of operator dot. Technical Report P0060R0, September 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0060r0.html.

[How10] Lee William Howes. *Indexed dependence metadata and its applications in software performance optimisation.* PhD thesis, Imperial College London, 2010. http://www.leehowes.com/files/HowesThesis2010.pdf.

[SIM16] Boost.SIMD: Portable SIMD computation library, 2016. https://github.com/NumScale/boost.simd.

[SR16] Bjarne Stroustrup and Gabriel Dos Reis. Operator dot wording. Technical Report P00252R0, February 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0252r0.pdf.

[SSC+15] Ben Sander, Greg Stoner, Siu-Chi Chan, Wen-Heng (Jack) Chung, and Robin Maffeo. A C++ compiler for heterogeneous computing. Technical Report P0069R0, September 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0069r0.pdf.

[WRRR16] Michael Wong, Andrew Richards, Maria Rovatsou, and Ruyman Reyes. Khronos's OpenCL SYCL to support heterogeneous devices for C++. Technical Report P00236R0, February 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf.